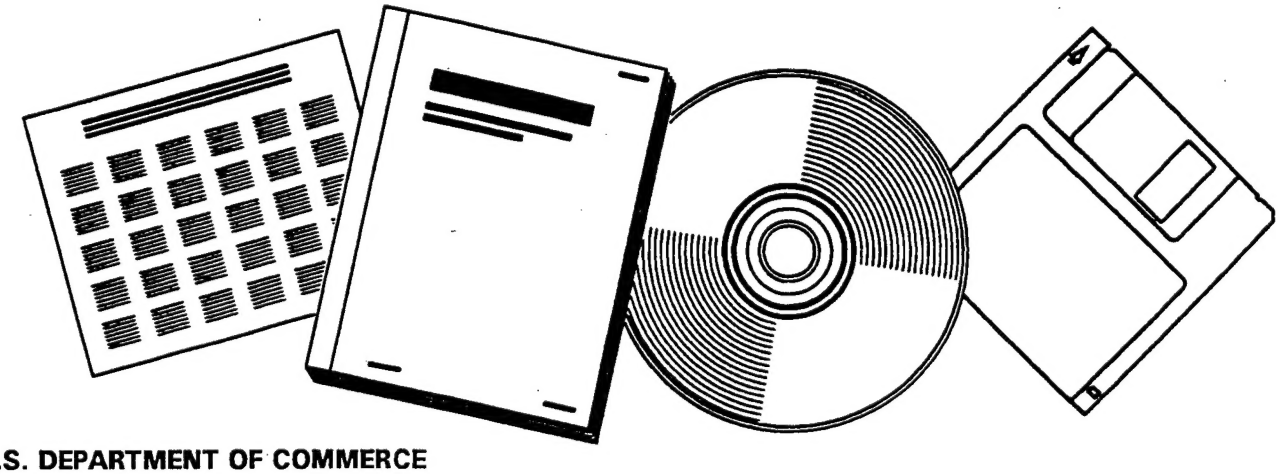
PB96-149760

PERFORMANCE AND FAULT-TOLERANCE IN A CACHE FOR DISTRIBUTED FILE SERVICE

STANFORD UNIV., CA

DEC 90



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

December 1990

Report No. STAN-CS-91-1363

thesis

PB96-149760

Performance and Fault-Tolerance in a Cache for Distributed File Service

by

Cary Gordon Gray

Department of Computer Science

Stanford University

Stanford, California 94305

PERFORMANCE AND FAULT-TOLERANCE IN A CACHE FOR DISTRIBUTED FILE SERVICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Cary Gordon Gray
December 1990

Abstract

Caching data on client workstations can improve the performance of file service in a distributed system. Distributed systems, though, are subject to host crashes and communication failures, and techniques that are commonly used to improve the performance of file caching do not tolerate these failures. This dissertation describes how high performance can be obtained from caching while still tolerating failures and providing at least the same level of coherence, availability, and reliability that the file service would have without caching.

Leasing is a time-based mechanism that guarantees that access to cached data is coherent with respect to arbitrary communication. The requirements for leasing to function correctly are stated, and it is shown how those requirements can be efficiently satisfied in a practical system that has both host crashes and lost messages. Leasing's performance is evaluated using both closed-form estimates and trace-driven simulation. For the pattern of access to files in V and similar systems, performance is very good for lease terms of just a few seconds; terms of this duration also give acceptable bounds on delays added due to failures. Several previous mechanisms for cache coherence are included as special cases of leasing, and leasing also allows a range of policies that can accommodate different access patterns. In addition, leasing can be extended to work with multi-level caching, with replicated data, and in conjunction with transactions.

A prototype file-service cache has been implemented for the V distributed operating system; the cache's design is described, and its performance is evaluated based on traces of file-system access. Three techniques enable the cache to reduce traffic by as much as 60% compared to a simpler cache: using leasing for coherence, distinguishing among a few classes of files, and caching information about files along with their contents. In contrast with other designs, the prototype achieves this reduction in traffic without sacrificing coherence, reliability, or availability.

Acknowledgements

This research was done within the Distributed Systems Group at Stanford University, headed by Professor David Cheriton; I would like to thank the many members of DSG in the years I was associated with it for making it an interesting place to work and for teaching me so much about computer systems and research. Special thanks are due to those who provided software that contributed directly to my work: William Lees, whose first implementation of the caching server, as a class project, provided a starting point for my work; Ed Sznyter, for his work on the V's virtual memory system; Ross Finlayson, Mike Nishimoto, Peter Brundrett, Paul Roy, and Tim Mann for their contributions to the file-server software.

The Andrew benchmark, on which one of the traces in Chapter 3 is based, was graciously provided by M. Satyanarayanan of Carnegie-Mellon University.

Portions of Chapter 2 were previously published, in revised form, as [30], and appear here by permission of the ACM; the material that appears here represents my own work.

During part of time at Stanford, I was supported by an NSF Graduate Fellowship and a TRW Fellowship Augmentation Grant. This work was also supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by National Science Foundation Grant DCR-83-52048, and by Digital Equipment Corporation.

Technical and financial support seem small compared to the moral support and friendship that kept me going as my doctoral work dragged on for far too long. Within DSG, I am especially grateful for both the encouragement and instruction I received from Marvin Theimer, Ross Finlayson, Bruce Hitson, Steve Deering, and Joe Pallas: without their encouragement I am sure I would have given up. Joe, in particular, endured far more interruption to his work than anyone else, and contributed more than I could ever repay. Gio Wiederhold and Jim Gray saw value in my work when I was unsure that I had anything worth saying. I am also grateful to all of my readers for their encouragement, wise suggestions, and quick response during the final push.

Much joy in my time at Stanford was provided by my extended family at church in Palo Alto, who put up with me, cared for me, and helped me keep everything in perspective. Joel Dager, Glenn Bates, and Howard Lin unwittingly kept me sane through some very difficult times. John Wilson's help and companionship during the actual writing were absolutely invaluable.

I am very grateful to my parents for their support and practical suggestions, and for their patience when mine was exhausted. My wife Emily endured my graduate studies dragging on for much longer than anticipated, and me being less than pleasant to live with during a significant portion of that time; I can not thank her enough.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

File storage is an important service in a general purpose computing system, whether that system is centralized or distributed. Files provide a basis for sharing data among users and applications as well as for preserving data over an extended period. Overall performance in many applications is limited by the performance of the file service.

Several trends in distributed systems are increasing the pressure on performance of distributed file systems. First, as systems grow to include hundreds or even thousands of hosts, a server needs to be able to support a larger number of clients. Second, as systems spread beyond local area networks, increased communication latency adds to response time. Finally, as processor speed increases, the delays for communication with a remote server and for access to secondary storage account for a larger fraction of the response time.

Caching file data on client workstations can significantly improve the performance and scalability of a distributed file service, as systems such as Andrew [35] and Sprite [50] have demonstrated. These systems, though, typically trade the robustness of the file service for additional performance gains from caching. Commonly used techniques for boosting cache performance sacrifice the coherence, reliability or availability of the file service, particularly in the presence of the partial failures that characterize distributed systems.

This dissertation focuses on the problem of obtaining caching's performance benefits without making these sacrifices. It presents *leasing*, a fault-tolerant mechanism for cache coherence, and describes a prototype file cache that uses leasing along with other techniques to provide both high performance and robustness.

1.1 Caching and performance

File-service performance is measured in terms of either throughput or response time. Throughput can be viewed in terms of capacity of shared resources: how many clients can a server or network support? Efforts to construct systems with very large numbers of clients call for supporting as many clients per server as possible, in order to minimize the number of servers for reasons of both cost and manageability.

Response time measures how long it takes to complete some task. The contribution of file operations to response time includes time for communication, for processing and device access at the server, and also for queueing when the server is congested. Response time also includes computation and possibly other activities in addition to file operations. For the systems examined by Lazowska, *et al.*, [40] communication delays were a very small fraction of total response time. However, when either communication latency increases, as on a wide-area network, or processors get faster, the delays for remote access to files can become a significant fraction of response time. Large numbers of clients add queueing delays as shared servers become congested, further increasing response time.

Caching data on clients can both increase server capacity and reduce response time. Server capacity increases because the use of cached data lowers the number of requests that a client makes of the server. Response time also benefits from the reduction in traffic: the client waits for a smaller number of synchronous operations to be performed by the server, and the queueing delays for those operations are also shorter.

1.2 Tolerating failures

Distributed systems experience failures: hosts can crash, and communications can be disrupted. The failures experienced by a distributed system are different from those in a centralized system. When the host crashes in a centralized system, all processing ceases. In a distributed system, however, a host crash is only a partial failure, and processing on other hosts continues, possibly unaware that any failure has occurred. A distributed system is also subject to communication failures: a message can be sent but never received. Lost messages and host crashes cannot always be distinguished, as either manifests itself as the failure to receive an expected response from another host.

In addition, these partial failures occur more frequently in a distributed system than do crashes of a centralized system, simply because there are more components that can

fail. Furthermore, workstations are commonly placed in relatively hostile environments—*e.g.*, offices, where the temperature and power supply are not controlled as well as in a machine room, and where users have ready access to switches and cables. Communication failures become more likely as a network grows in terms of either distance or number of hosts. Failures must be expected in any distributed system of nontrivial size.

A robust distributed system must tolerate these failures: parts of the system that are not directly affected by a failure should continue to function correctly. For a distributed file service, tolerating failures requires preserving three properties: availability, coherence, and reliability.

Availability. A partial failure should not unnecessarily make file data unavailable to the remaining system. A client should be able to access a file whenever it can communicate with the server that stores the file.[1] Because client failures are more common, the failure of one client should not make any file unaccessible to other clients.

Coherence. To facilitate sharing, access to file data should be *coherent*. Informally, coherence implies the same behavior as in the absence of caches: reading a file returns the data most recently written to it. The problem arises when a file that is already in one client's cache is written by another client. The cached copy now differs from that stored by the file server, and the system must guarantee that subsequent reads return the server's data instead of that in the cache.[2] Coherence should be preserved in spite of failures.

Reliability. The possibility of partial failures makes the reliability of the file service more important. Because a partial failure may not be noticed by a user or application, confusion can result if the failure causes data to be lost. Users and applications need the assurance that both the data they write and the data they read will persist in spite of noncatastrophic failures.[3]

To illustrate the need for reliability, let us consider the policy of *delayed write* employed in many centralized Unix systems [66], in which newly written file data is allowed

[1] Or, for a replicated file, when it can communicate with the required set of servers.

[2] A more precise definition of coherence is developed in Chapter 2. At this point, please note that coherence is a basic property expressed in terms of individual operations, each on a single data item; coherence should not be confused with, for example, database integrity, which is expressed in terms of groups of operations (transactions) and the relationships among the values of multiple items.

[3] Host crashes and lost messages are anticipated; other types of failures, such as a disk head crash, might still be regarded as catastrophic and result in loss of data.

to remain in volatile memory for a limited time before it is written to disk. Delayed write improves performance in two ways. First, it reduces the demand on disks, because a significant fraction of file data is short-lived: if a block is rewritten or a file deleted within the delay period, then a write to disk can be avoided. Second, it lowers response time, because a write operation returns as soon as the data has been copied into a file-system buffer, without having to wait for a slow disk write.

Delayed write introduces the risk of losing recently written data if the system crashes. The potential loss is therefore limited by periodically writing all dirty buffers to disk, typically at an interval of thirty seconds. In a centralized system, it is unlikely that a user fails to observe a failure in which he loses data. The crash must occur within thirty seconds of the lost write, which means a user may unknowingly lose only data written within thirty seconds before logging out, or written by background processes while the user is not logged in. Any program processing the (soon-to-be) lost data is terminated by the crash; any writes it performed that depended on the lost data are younger than the delay interval and so are probably lost as well. Manual intervention can be used to recover for simple interactive activities, particularly when the user is a programmer.

In a distributed system, it is more likely that a failure goes undetected by a particular user. If commands are remotely executed at hosts selected by the system (as in [64]), a user will not notice if a remote host crashes immediately after he executes a command there. Similarly, a user will not notice the failure of a file server if it recovers before he next uses it. A user that is not aware that data has been lost will not take appropriate action to recover from the loss.

The problem of undetected failures is worse for programs than for human users. Consider a program that reads from one file, then writes to another, with the requirement that the output file always reflect some prior state of the input. If the program reads newly written data from the input file shortly before the server storing it crashes, losing the input data, the program can continue to execute and successfully write an inconsistent version of the output file on a different server.[4] Such possibilities make it difficult to construct robust applications.

[4] Or the same server, if the computation is long enough for the server to recover.

1.3 Contributions of this dissertation

This dissertation examines the use of caching in providing a distributed file service that is scalable and high-performance, and that tolerates common failures without compromising the coherence, availability, or reliability of file storage.

Chapter 2 addressed the problem of efficiently maintaining the coherence of caches in the face of failures. It describes *leasing* as a solution to this problem, and states requirements for leasing to function correctly in spite of partial failures. A simple analytical model for leasing's performance is developed, and that model shows that leasing can perform quite well for file-service access patterns while still giving acceptable failure-case behavior. The conditions for correctness allow a range of policies for managing leases, so that leasing can accommodate other access patterns as well.

A prototype file-service cache that has been built for the V distributed system is described and evaluated in Chapter 3. The evaluation is in terms of server traffic, based on analyzing traces of file-system access collected in the V-system. Three enhancements allow the prototype to reduce traffic without losing robustness: distinguishing temporary from other files, caching information about files, as well as their contents, and using leasing for coherence.

Chapter 4 explores issues beyond the scope of the prototype. It describes how leasing can support multi-level caches in systems of very large scale, and it examines how leasing could be used for caching in a storage service based on transactions or with replicated data.

Chapter 2

Coherence

Intuitively, caches are coherent[1] if each read of a data item returns the value most recently written to that item. Coherence implies that the results of operations on files are not affected by the use of caching in the system. This property is important because it keeps the model of file service simple, while allowing data to be shared via files.

Existing approaches to ensuring coherence have one of three shortcomings. Some methods depend on reliable communication, and so do not tolerate failures. Other approaches require a coherence check on each read access, and so do not perform well. Finally, some approaches work only by forbidding all updates to cacheable data, and so are of limited applicability.

This chapter describes and evaluates *leasing*, a mechanism for coherence that tolerates both site and communications failures. The first section explores the problem of coherence in greater detail and identifies the aspects of the problem that are the focus of the remainder of the chapter. Section 2.2 describes leasing, and Section 2.3 analyzes its performance for distributed file service. Finally, Section 2.4 examines several extensions to the basic mechanism.

2.1 Background

The potential for incoherent access arises from the fact that caching a data item creates an additional copy of it. If the cached copy comes to differ in value from the one stored

[1]To minimize confusion, we use the term *coherence* for the property that is also commonly labelled *consistency* in descriptions of file systems or memory systems. Unfortunately, *consistency* denotes a different concept in the literature on databases (*e.g.*, [21, 32]), and the term has been used somewhat inconsistently in the literature on file caching (*e.g.*, [36, 50, 59]). See section 2.1.3 for more details.

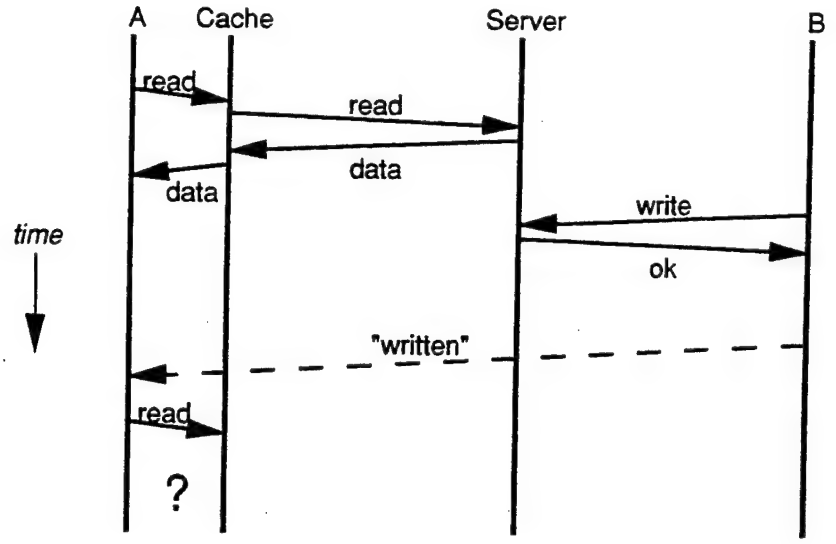


Figure 2.1: Potentially incoherent access.

by the file server, reading from the cached copy can produce anomalous results. For example, consider the events depicted in Figure 2.1. User (or application) A reads from a file via the cache on one workstation, and then B writes to the same file from some other host. B somehow informs A of the write, so that when A next reads from the file, A expects to see what B wrote. The system must somehow ensure that A's final read returns the latest value instead of the old version in the cache.

2.1.1 Defining coherence

The most commonly given definition is that a system is coherent if each read of an item returns the value most recently written to that item (*e.g.*, [12, 42]). In a distributed system, though, events are not totally ordered [38], so that "most recently" is not well-defined, and a more precise definition is needed.

What coherence requires is that the results of a set of operations performed with caches not differ detectably from what they would be if there were only one copy of each item—*i.e.*, if there were no caches. A precise definition of coherence is based on two partial orderings of operations in a system's execution. One ordering is based on observable events, such as the sending or receiving of a message, or an agent performing a series of actions sequentially [38]. In Figure 2.1, the "written" message from B to A allows A to observe that B's write occurs before the final read is requested. The second partial order, which we call the *version order* is determined by the results of operations: each read operation returns a value from a write operation that immediately precedes

it in the version order, and the read precedes the operation which overwrites the value. Whenever these two orders conflict, it appears that an operation has been performed out-of-order. We therefore describe a file service as coherent *with respect to* a particular set of observable orderings on operations if the observable and version orderings produced are always compatible.[2]

There are several possible choices for the set of observable orderings. One possibility is to limit observations to reading and writing files; another is to consider only messages sent within the computer system's communications network. If either of these standards is chosen, however, access will not appear coherent to users who communicate by other means. In practice users do engage in such communication: users converse with each other, or a single user may employ more than one host. Consequently, a distributed file service needs to ensure that access is coherent with respect to arbitrary communication, not just communication via files or via the distributed system's network. Within the rest of this dissertation, therefore, we use "coherence" as shorthand for "coherence with respect to arbitrary communication."

2.1.2 What are reads and writes?

Within this chapter we consider a very simple abstract file service in which each operation is either a read or a write of a single data item, and individual read and write operations are performed atomically.[3] In relating the operations of an actual file system to this model, those factors must be considered. First, other operations can be viewed as a sequence of reads and writes that is performed in a batch by the server. Second, the operations that correspond to read and write from the standpoint of coherence are determined by the visibility of those operations with respect to each other. For example, in Unix systems, the `read()` and `write()` system calls are reads and writes from the standpoint of coherence because the effects of a `write()` are visible to any subsequent `read()` of the same data.[4] In the V file system, however, opening a file for reading gives access to a snapshot of its state at the time of the open, and writes to an open file become visible to subsequent opens only after the file is committed (usually by closing it). For the purposes of coherence, then, read and write in V correspond to file open and commit.

We also need to distinguish the atomic event of performing an operation from among the series of events involved in processing it. Each operation comprises at least four

[2]More formally, if the directed graph of their union is acyclic.

[3]This abstraction is also a common model for a computer's memory system.

[4]Because of buffering, the same is not true for the `fread()` and `fwrite()` library calls.

communication events, the sending and receiving of each of the request and result, in addition to some sequence of processing by the server or cache. An operation is performed by whatever step in the processing both determines its result and allows that result to be visible. For a write, in particular, the result is first made visible either when the server returns an indication of success to the writer or when the written value is returned to a reader. It is possible for a server to record a write to disk yet, for a reason such as maintaining coherence, not make it visible for some time thereafter. In such a case performing the write is tied to its visibility, rather than some other point in its processing.

2.1.3 Related concepts

Coherence is a primitive property expressed in terms of individual operations, each on a single data item; it should not be confused with more sophisticated constraints in terms of multiple items or multiple operations. Database integrity, for example, can require that relationships be maintained among the values of multiple items. Similarly, serializability of transactions imposes a logical ordering on groups of operations. Enforcing properties such as multi-item integrity constraints or serializability of transactions requires additional mechanisms. Coherence guarantees only that the results of operations are the same as they would be if there were only one copy of each data item.

2.1.4 Restrictions

For simplicity, we limit our description in the next section to caches for which all writes go through to the file server and to nonvolatile storage, so that newly written data becomes visible and persistent at the same time. This property simplifies the writing of robust applications, because a program can assume that any data it reads will not be lost due to a host crash.[5] Coherence is simplified for write-through caches because all write operations are performed synchronously by the server, ensuring that clients already see a consistent ordering of writes. Write-back caches do not involve the server in handling each write request, so that the coherence mechanism for such caches must guarantee their order. Section 2.4.3 describes how leasing can provide coherence for caches that are not write-through, which may be appropriate in contexts other than plain file service.

[5]Some consider write-through to be prohibitively expensive for file service. In the V-system, however, writes become visible to other programs only when the writer closes or explicitly commits the file, so that write-through applies only to close and commit operations. Section 3.7 considers how the cost of write-through can be minimized in other systems.

The analysis of performance in Section 2.3 is also oriented toward a simple file service. In addition to assuming write-through caches, it focuses on the cases for operation rates and lease terms characteristic of file access in a workstation-based system such as V.

2.2 Leasing

A *lease* is a contract that gives its holder specified rights over property for a limited period of time. In the context of caching, a lease grants to its holder authority over writes to the covered data item during the *term* of the lease, such that the grantor must obtain approval from the leaseholder before allowing the item to be written. When a cache fetches data from a file, it also obtains from the server a lease that covers the data; that lease prohibits any write to that data during the lease's *term* unless the server first obtains the approval of the cache. When a client writes to a file, the server requests approval of the write from the holders of all unexpired leases covering the written data: the write must be delayed until all leases have either expired or had approval granted. Before it grants approval of a write, the cache invalidates its local copy of the data.

When the file is read from again within the term of the lease, the cache provides immediate access to the file without communicating with the server. After the lease expires, however, the cache has no assurance that the file has not been written; the cache therefore queries the server to learn of writes occurring since its lease expired and to obtain an extension of the lease.

Figure 2.2 provides an example of how leasing works. As in the previous example, A reads a file via a cache, and the cache fetches the data from the server. In addition to the data, however, the cache obtains a lease over the file, which allows a subsequent read during the lease's term to be done without communicating with the server. After the lease expires, the cache requests an extension, which the server grants.

When B attempts to write the file during the term of the lease, the server requests the approval of the leaseholder, A, and delays performing the write until A's approval is received. B's final write request, received after the lease has expired, is processed without having to obtain approval from A.

2.2.1 Conditions for correctness

A pair of conditions is sufficient to ensure that reads return coherent data, and these conditions divide responsibility between server and client. First, the server must honor

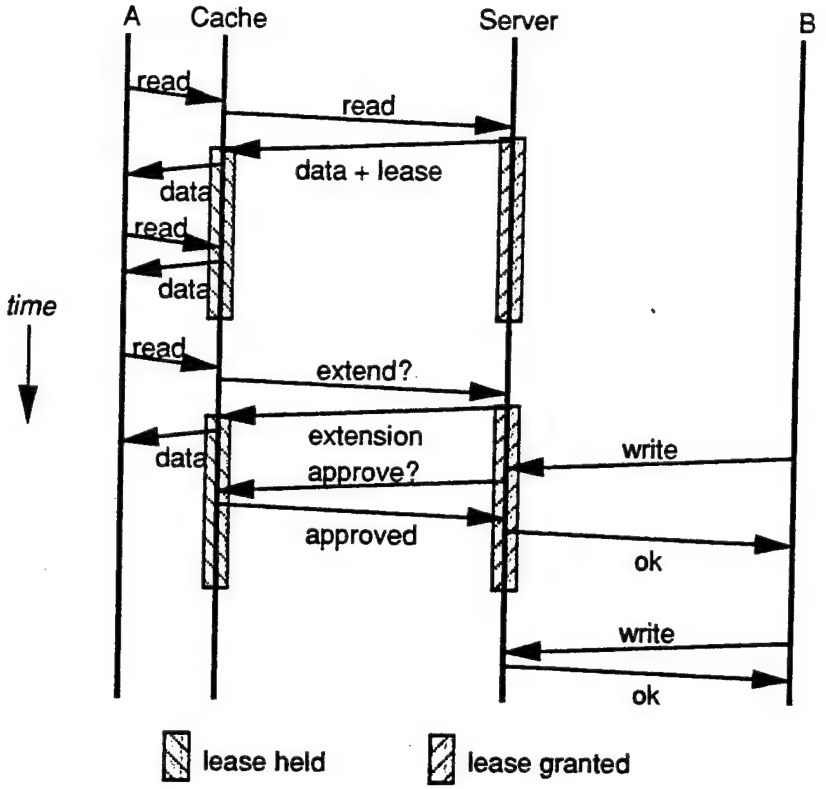


Figure 2.2: An example of caching using leasing.

the leases it grants:

> The server must not perform a write of an item while there exists an unexpired lease covering it whose holder has not approved the write.

On the client's side, the cache knows data is current only if it holds a lease over it when it is read:

> A cache may return its copy of a data item in response to a read only if it holds a lease over the item whose term includes the interval from the time the item was fetched or validated to some time after the read request is received, and the cache has not approved a write during that interval.

These conditions neatly partition responsibility between the client and server, with the exception of one detail. While most events, such as the granting of a lease or approval of a write, involve exchanging a message and so are ordered at both sites, the expiration of a lease is not: instead, cache and server must measure the term with imperfect local clocks. If the clocks are synchronized to differ by at most ϵ, the term can be expressed as ending when the server clock reads T, and the cache may safely return the covered data

as long as the client clock reads less than $T - \epsilon$. Without synchronized clocks, the term is communicated as its duration t, and the client makes a somewhat larger allowance for the (bounded) relative drift of the timers and for communication delay. The minimum required is that the client clock not run "too slow" relative to the server clock.

2.2.2 Coping with failures

Each client's coherence depends only on itself and the server. No loss of messages can compromise correctness, because the correctness conditions for both the cache and the server can be satisfied without communication. The server can satisfy its condition by delaying a write until leases expire. The cache's condition is met as long as no reads return cached data; reading cached data is therefore allowed only during the term of a previously obtained lease and when the cache is able to communicate with the server.

The impact on one client of another client's failure is limited to possibly increased delay for writes, but only until the failed client's leases have expired. That delay can be bounded by the server by limiting the terms of the leases it grants. In addition, the server needs to avoid starvation of writes: while a write is waiting for approval or for leases to expire, the server limits the terms of new or extended leases. The server can thus guarantee a maximum acceptable delay for writes in the event of failure of a component not critical to the operation, simply by limiting the terms of leases.

The server must honor the leases it grants even across crashes; some record of leases must therefore be kept on non-volatile storage. However, both the volume of data kept and frequency with which it is written can be reduced by observing that only an upper bound on when leases expire is required to ensure correctness. In the extreme case, the server could store just a maximum term and then delay any write requests during that interval following recovery.[6] The expense of maintaining a more detailed record in order to reduce the hold-down interval is probably not justified in most cases, since the added delay is no worse than that caused by failure of a client. For the terms we examine in Section 2.3.2 the hold-down is not an odious constraint.

Concern for availability, then gives us a constraint on how long a term may be: the term is an upper bound on the delay coherence can introduce in the event of failure pertaining to another client, and that delay is experienced only on writes. A sufficiently short term also allows reduction in the amount of nonvolatile bookkeeping required of

[6] If a minimum time to recover is known, that interval may be subtracted from the hold-down period. For many systems, in fact, we expect that the time to recover will exceed the maximum term, in which case no hold-down is required.

the server, at the expense of the same added delay for writes after a server crash.

2.2.3 Other coherence mechanisms as special cases

Several of the mechanisms previously used for coherence in distributed file systems can be expressed as special cases of leasing. Schemes that require a check on each use of cached data (as in, for example, the first version of the Andrew file system [58]) correspond to a term of zero duration; they incur a high level of overhead for extensions, but add no delay to writes in the event of a failure. Mechanisms that depend on reliably notifying caches of writes (as in the later version of Andrew [35]) correspond to an infinite term, in which case coherence can be preserved only at the cost of unbounded delay—effectively unavailability—for writes in the event of a failure. The caching of only immutable data (as in the Cedar file system [60]) also corresponds to an infinite term, but because writes are not supported, it causes no problems.

2.3 Performance

This section examines the performance of leasing, focusing especially on how the choice of term affects performance. We have two goals for performance: high system throughput and low application response time. Throughput is limited by the demands on bottleneck resources, which previous studies [40, 50] have identified as the server CPU and, to a lesser degree, the network. To a first approximation, the impact of coherence on both throughput and response time can be evaluated in terms of message traffic: the demand on the server and network is roughly proportional to the number of messages they handle, and the delay added to operations is dominated by the latency for synchronous message exchanges.

The first subsection develops a simple analytical model for predicting the contributions of leasing to load and delay; the next subsection applies this model to measurements of file-system access in V and compares its predictions with results from trace simulation. Both the predictions and simulation results show that leasing with terms of just a few seconds performs quite well for V's file service, and that it continues to perform well when the system is extended with faster processors and wide-area networks. The final subsection examines several issues of lease management and how they affect sharing.

2.3.1 Analytical model

Performance depends on several aspects of the system and the pattern of access to files. We consider the simple case of a single server with N clients, where each client's reads and writes follow Poisson distributions with rates R and W, respectively,[7] and all of the files at each client are covered by a single lease. Each file written is shared by S caches.

The time for communication is split into two components, processing and propagation, so that we can model a simple multicast facility [17]. Multicast messages are sent once, and received with high probability by the intended recipients. A message requires m_{proc} seconds of processing at both the sender and recipient, plus m_{prop} seconds for propagation between them; so a message is received $m_{prop} + 2m_{proc}$ after it is sent. (These averages include a normal level of retransmissions.) A unicast request and reply therefore takes twice that time, $2m_{prop} + 4m_{proc}$. The time required to send a multicast message and collect replies from its n recipients is denoted by $m_{multi}(n)$. For small n, such that the effects of congestion and bookkeeping overhead are not large, this value is approximately one round-trip time, to receive the first reply, plus for the sender to process the additional $n - 1$ replies, giving

$$m_{multi}(n) = 2m_{prop} + (n + 3)m_{proc}.$$

For a lease with term t_S, the effective term t_C during which the cache *knows* it holds an unexpired lease is shortened by the time for the cache to learn of the lease as well as an allowance ϵ for imperfect clocks. Thus, this effective term is given by

$$t_C = \max(0, t_S - (m_{prop} + 2m_{proc}) - \epsilon).$$

A term of less than $m_{prop} + 2m_{proc} + \epsilon$ cannot improve performance over a term of zero.

With these parameters, summarized in Table 2.1, we can derive estimates of performance in the absence of failures.[8] The analysis is simplified by ignoring queueing delays due to congestion as well as the second-order effect of response time on request rate.

The message traffic for coherence has two components: that for supporting reads (by extending leases) and that for obtaining approval of writes.

A cache requests an extension when it receives a read request and its lease has expired; it receives an expected Rt_C additional reads before the lease expires. The pair of messages

[7] Realistically, one would expect that both reads and writes would be clustered to a greater degree than is represented in a Poisson distribution. As noted in section 2.3.2, this makes the estimates of performance slightly pessimistic.

[8] Recall that the delay added by coherence in the event of a failure is bounded by the maximum term of leases granted, and that only writes incur that delay.

Symbol	Description
N	number of clients (caches)
R	rate of reads for each client
W	rate of writes for each client
S	number of caches in which a file is shared
m_{prop}	propagation delay for a message
m_{proc}	time to process a message (send or receive)
$m_{multi}(n)$	time to multicast a message and collect replies from n recipients
ϵ	allowance for uncertainty in clocks
t_S	lease term (at server)
t_C	effective lease term at cache, $\max(0, t_S - (m_{prop} + 2m_{proc}) - \epsilon)$
$t_a(S)$	time to obtain approval from S leaseholders

Table 2.1: Performance model parameters.

to request and grant the extension are amortized over $1 + Rt_C$ reads, so that the rate of extension-related messages handled by the server is

$$x_{ext} = \frac{2NR}{1 + Rt_C}$$

and an average delay of

$$d_{read} = \frac{2(m_{prop} + 2m_{proc})}{1 + Rt_C}$$

is added to each read request.

To get approval for a write, the server multicasts the request for approval and processes the replies from all of the leaseholders. When the writer is one of the leaseholders, one approval message can be saved if the request for a write carries the implicit approval of the requesting cache. For the common case of writes to files that are not shared, the implicit approval eliminates approvals altogether. For a shared file, obtaining approval requires the one multicast plus $S - 1$ approvals, for a total of S messages,[9] and the time $t_a(S)$ to gain approval is $m_{multi}(S - 1)$ for $S > 1$. There is benefit to seeking approval only if the term remaining in the leases exceeds $t_a(S)$; otherwise both delay and traffic are lower if the server simply waits for the leases to expire. If the server term t_S is less than $t_a(S)$, then approval is never sought, giving write traffic of zero, and a delay of at most t_S. For larger t_S, the delay is at most $t_a(S)$ and the traffic at most NSW. The

[9]Without multicast, each approval requires $2(S - 1)$ messages.

actual traffic should be smaller, since when a write is requested it is likely that some of the leases have expired or are closer than $t_a(S)$ to expiring, so that S approvals are not always required. The difference is significant, however, only when $t_a(S)$ is a significant fraction of the term or when the total delay is dominated by that for approval of writes.

When the two components are combined, there emerge two important thresholds for the term. The first threshold is the point at which t_C becomes non-zero; below this point writes are penalized but reads do not benefit. The other threshold is when $t_S = t_a(S)$, below which traffic for writes is zero and delay increases linearly, and above which they grow to NSW and $t_a(S)$, respectively. For file caching, the terms of primary interest are much larger than either of these thresholds. For such terms the server's total traffic is approximately

$$x_{total} = \frac{2NR}{1 + Rt_C} + NSW$$

coherence-related messages per unit time, and the average delay that coherence adds to each read or write is

$$d_{avg} = \frac{1}{R + W}\left(\frac{2R(m_{prop} + 2m_{proc})}{1 + Rt_C} + Wt_a(S)\right).$$

The minimum total traffic is always either $2NR$, for a term of zero, or NSW, for an infinite term. To bound delay due to failures, though, the terms must be limited to fairly short values, which raises the question of whether an acceptably short term will reduce traffic below that for zero. For a term longer than $t_a(S)$, this is true if

$$2NR > \frac{2NR}{1 + Rt_C} + NSW.$$

Defining a *lease benefit factor* as

$$\alpha = \frac{2R}{SW},$$

the preceding condition holds if $\alpha > 1$ and

$$t_C > \frac{1}{R(\alpha - 1)}.$$

A sufficiently long lease term will reduce server traffic whenever α is greater than one, and larger values of α and R imply better performance for short terms.[10]

[10]Without multicast, the total number of approval related messages is $2N(S-1)W$, and $\alpha = R/(S-1)W$.

rate of reads	R	1.44 /sec
rate of writes	W	0.0399 /sec
message propagation time	m_{prop}	1.0 msec
message processing time	m_{proc}	0.25 msec
allowance for clocks	ϵ	100 msec

Table 2.2: Parameters for file caching in V.

2.3.2 V file service

This section applies the preceding analysis to data from the V-System as a concrete example. Comparing the results with those from trace simulation provides some validation for the model. The analysis indicates that terms of just a few seconds perform quite well, while limiting the impact of failures, and that they continue to perform well with increases in network latency or processor speed.

The parameters for the analysis are taken from the `fsbuild` trace of file access, which is described more fully in the next chapter and in Appendix A. The measured rates of reads and writes[11] are given in Table 2.2, along with estimated times for communication. The trace is of a single client, so that it contains no sharing.

Figure 2.3 shows the traffic generated by coherence as a function of the term, relative to the traffic for a zero term—*i.e.*, a check on each read access. The curve labelled *trace* is the result of trace-driven simulation, while the others are from the analytical estimates, with the various levels of sharing indicated. The *trace* and $S = 1$ curve are both for the no-sharing case, and so should match; they are close, with *trace* showing lower traffic for all but the very shortest terms. The estimated value is high because because actual traffic is burstier than is reflected by a Poisson distribution; this burstiness gives the *trace* curve a sharper and lower knee. The cross-over for terms near zero is due to two factors: first, there is in reality a minimum time between operations of a few milliseconds, which the Poisson distribution does not capture, and, second, the timestamps in the trace have a granularity of ten milliseconds, which amplifies the difference between the measured and Poisson interarrival times. Overall, though, the estimate produces reasonable results, and can be expected to slightly overestimate traffic for terms in this range.

Coherence traffic drops off quickly from its peak for short terms, then flattens out.

[11] These are rates of operations that count as a read or write from the standpoint of coherence, not block-level accesses. Additional information about the V file system can be found in Section 3.1 of the next chapter.

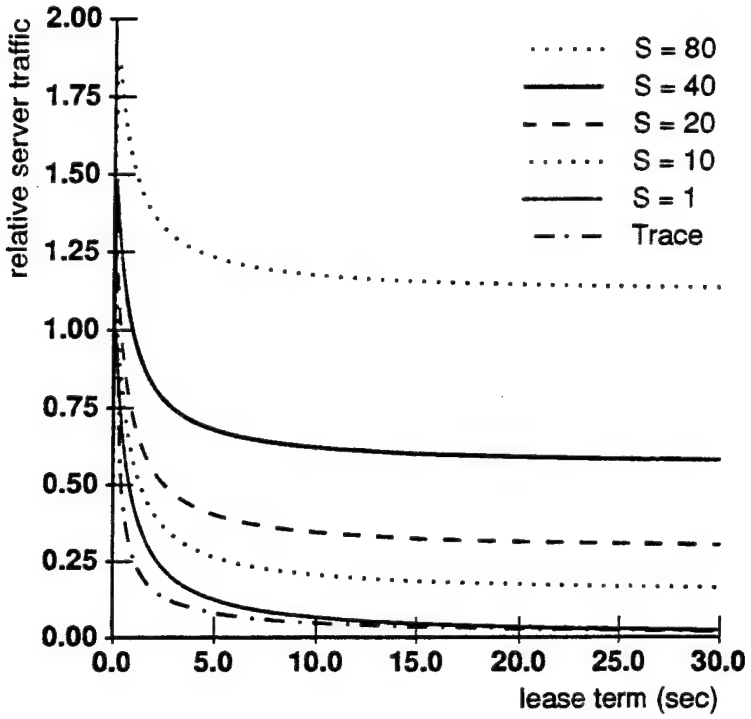


Figure 2.3: Traffic for coherence in V.

For example, a term of ten seconds produces about 5% as much coherence traffic as a zero term, while a thirty second term reaches 2%. While there is a 60% drop in coherence traffic between ten and thirty seconds, the server is handling enough other traffic that the overall gain is much smaller. In the measured system, coherence accounts for 43% of the all server traffic when the term is zero, so that increasing the term from zero to ten seconds reduces overall traffic by more than 40%. The increase from ten to thirty seconds, though, achieves a further drop of less than 2%. Recalling that the term is the maximum delay that coherence can add to a write, and so is bounded by the acceptable increase in delay in the event of a failure, terms of up to ten seconds seem reasonable, and they are very effective at reducing traffic.

The other curves in Figure 2.3 give some idea of how sharing affects traffic. As for the no-sharing case, the bulk of the gain from increasing the term comes for moderately short terms. For higher degrees of sharing, though, the added traffic for approval of writes limits the reductions. As the level of sharing increases, the minimum term to reduce traffic below that for a zero term increases, and for a high enough level of sharing, no reduction is possible.[12]

[12]Realistically, our estimates for time and traffic to gain approval break down before the degree of

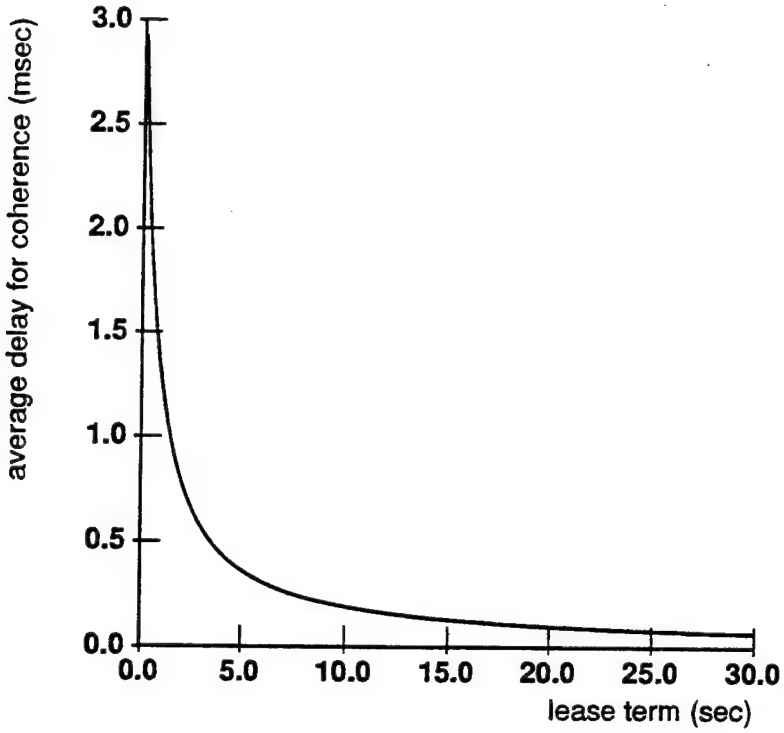


Figure 2.4: Average delay for coherence in V.

The contribution of coherence to delay for each operation is shown in Figure 2.4. The curves for different degrees of sharing are almost indistinguishable at the scale shown; so only the curve for $S = 1$ is included. For a term of zero, a round-trip time of 2.5 milliseconds is added to each read; for longer terms, the delay falls of sharply, in the same manner as the traffic. In the system measured, the application does 677 milliseconds of other work for each read or write performed,[13] so that even at 2.5 milliseconds per operation the contribution of coherence is quite small. This matches the findings of Lazowska, *et al.*, [40], who report that in the absence of contention and on processors of modest speed, the contribution to response time of even uncached remote file access is fairly small fraction of the total. The major benefit of leasing in this case is the reduction in server traffic, and with it a reduction in contention for server resources.

The results presented so far are for a moderately slow processor (MicroVAX II) on a local-area network, such that network latency is dwarfed by processing time; we now

sharing reaches twenty, because of contention for the network and for buffers; Danzig [18] derives much more complicated estimates that do consider contention. The extant data on sharing (*e.g.*, [49]) indicates, however. that most sharing falls into one of two categories: files with a significant level of writing are usually shared at very low degree, and files that are universally shared are very infrequently written.

[13]Recall that these reads and writes in V correspond to file open and close, not block level operations.

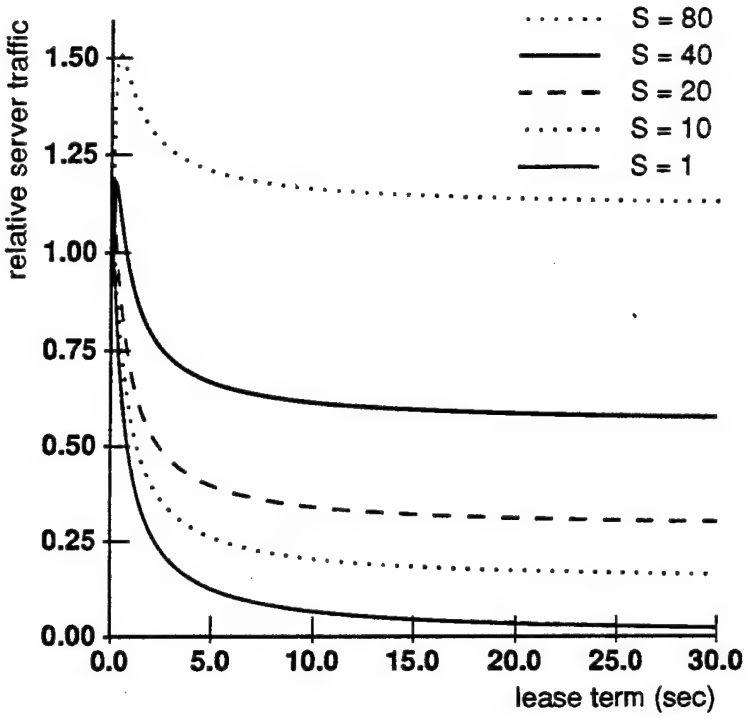


Figure 2.5: Traffic for coherence with 50 msec network latency.

consider how leasing affects performance on a higher-latency network or with faster processors, such that this is no longer the case.

Figure 2.5 shows server traffic with all parameters as before, except for the message latency m_{prop}, which is increased to 50 milliseconds, as might be seen on a wide-area network. The only significant change is that the traffic for the higher degrees of sharing has a lower peak for short terms. Traffic for approvals is reduced because the time to obtain approval $t_a(S)$ is longer, so that the server more often forgoes seeking approval in favor of simply waiting for less that $t_a(S)$.

The effect on delay is much greater, as shown in Figure 2.6: though the curve is qualitatively similar, the absolute delay per operation reflects the much higher round-trip time of 100 milliseconds. A zero term adds a round-trip time to each operation; in this case the almost 100 milliseconds per 677 milliseconds of computation increases response time by almost 15%. At a term of ten seconds, the added delay has fallen to 6.5 milliseconds per operation, or only about 1%, So modest terms for leases continue to perform well at the sort of latency expected for a high-performance wide-area network.

When processor speed is increased, the rates of operations increase, which gives the curves sharper and lower knees. In terms of traffic and added delay, fairly short terms

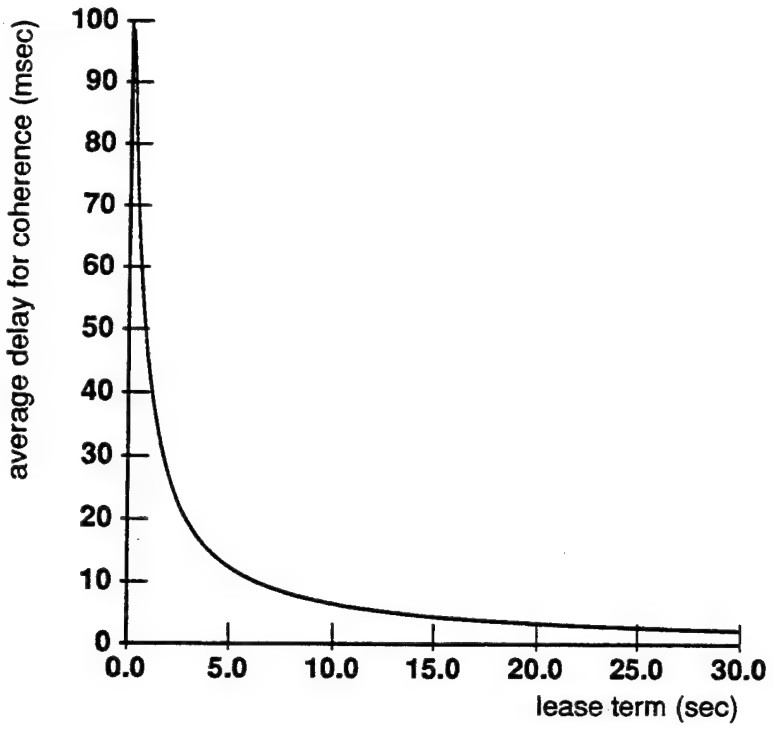


Figure 2.6: Average delay for coherence with 50 msec network latency.

should therefore perform even better. However, as the average time between operations decreases, the relative contribution of delay for coherence to response time increases. For example, if a faster processor increases the operation rate by a factor of five,[14] the time between operations falls to 135 milliseconds. On the local area network considered earlier, communication delays are still small compared to other computation. Delay is much more significant on the higher-latency network: a term of zero adds 100 milliseconds to the 135 millisecond response time, or nearly 75%. A term of ten seconds, though, brings the contribution of coherence down to 1.4 milliseconds per operation, or slightly over one percent.

In summary, the excellent performance for leasing with modest terms extends to faster processors and to higher-latency networks, and leasing has a larger impact on response time. Terms of on the order of ten seconds yield a significant reduction in total server traffic compared to a zero term (check-on-use), while longer terms provide little additional improvement. On local-area networks or with moderately slow processors, response time

[14]The time between operations is spent in more than just processing; it includes some time for I/O and communication, at lease for cache misses. A five-fold speedup therefore requires a *much* greater than five-fold increase in processor speed.

	public	private	combined
reads/sec	0.475	0.962	1.437
writes/sec	0	0.399	0.399
extensions/sec (est.)	0.0833	0.0914	0.0944
extensions/sec (trace)	0.0529	0.0496	0.0675

Table 2.3: Effect of separating extensions for file groups.

is dominated by other computation, so that the delays for coherence do not significantly contribute to response time. As either network latency or processor speed increases, though, the message round-trip time becomes significant compared to the amount of processing per operation, and the on-use coherence checks required by a zero lease term become a significant contributor to response time. Under these conditions, fairly short terms offer a significant improvement in response time.

2.3.3 Sharing, granularity, and multiple leases

The preceding analysis appears to assume that each client holds a single lease over all data its at the server, which raises the issues of the granularity at which leases are specified and the extent to which they are grouped for management. It also suggests the question of how performance is affected if a client's files are split between multiple servers.

From the standpoint of minimizing both traffic and delay, a client should seek to amortize each communication with the server over as many operations—and, therefore, as many items—as possible. Whenever it must request any extension from a server, the cache should request extension of all leases it holds from that server. Because traffic is bursty, doing so significantly reduces traffic. The files in the previously analyzed trace can be grouped by whether they are in the public or private portion of the name space. Table 2.3 shows the operation rates for these two groups; it adds the rate of lease extensions with a ten second term, figured by both the analytical estimates and trace simulation. The sum of the rates for separate extensions is nearly twice that for when extensions for both groups are combined. So when extensions can be combined, doing so significantly reduces the coherence traffic. If they cannot be combined, however, as when the two groups are stored on different servers, the total coherence traffic is larger, but still a small share of total file-server traffic.

The question of granularity in specifying leases is equivalent to the previously studied

problem of lock granularity in database systems [56]. A fine grain increases overhead, in memory, network bandwidth, and processing. A coarse grain increases contention, which in leasing takes the form of *false sharing* when a cache holds a lease over an item of which it has no copy.

False sharing also results when a cache holds a lease over data that the client is no longer using. One cause of this is that the lease term may extend beyond the period of interest, which provides additional motivation for keeping terms short. Another cause is the grouping of extension requests recommended above. To minimize false sharing, a cache should relinquish its lease when sharing is detected for items in which it is no longer interested; *i.e.*, when a cache receives a request to approve a write to an item that has not been accessed recently, the cache should relinquish its lease of that item (and any related items) instead of granting approval while retaining the lease.

2.4 Additional considerations

2.4.1 Options for lease management

Leases ensure coherence as long as the server and cache satisfy the conditions in Section 2.2.1, and these conditions leave considerable flexibility. In addition to initially setting the term, the server is free to extend a lease's term, and even to send an unsolicited message to the leaseholder informing it of the extension. Also, the server is not *required* to seek approval for a write; it can choose instead to wait for the lease to expire. When the number of leaseholders is high or it is probable that some leaseholder has failed, such a lazy approach to writes may perform better than attempting to acquire approvals. While waiting, the server can write the new data to disk, provided it ensures that it does not become visible before the lease expires.

The client controls a different set of options. For example, it chooses when to obtain and relinquish leases, in the same way that it chooses which data to retain in its cache. In addition, the client decides when to request that a lease be extended: when it expires, to avoid delay for read operations, or when the covered data is next used, to reduce the load on the server. One virtue of the latter option is that it imposes no load when a client is idle. If, however, the client requests extension whenever the lease is about to expire, reads incur no delay for coherence, since the lease does not expire; this reduction in delay is purchased at the cost of higher server traffic and insensitivity to changes in client activity. The client can do any of three things when it receives a request to approve

a write: it can grant approval, retaining the lease; it can allow the write to proceed by relinquishing its lease; or it can ignore the request, forcing the server to wait for lease to expire.

An example: installed files

As an example of how this flexibility can be exploited, consider publicly *installed* files, such as libraries and commands. These files account for a significant portion of access, but that access is almost exclusively reads; writes to installed files do occur, but only infrequently, and then typically encapsulated within some sort of explicit installation procedure. Installed files also account for most sharing, especially universal sharing: as the system grows larger, these files would be expected to be shared by some constant fraction of the clients, so that the number of leaseholders grows with the number of clients.

When the number of clients is large, the few write operations that are requested perform very poorly. A single request for approval elicits a reply from each of the clients, and the resulting implosion of messages to the server increases the likelihood that some of the approvals are lost due to either network congestion or buffer overruns [18]. Retransmissions increase both the delay and the traffic on the server well above those estimated in our performance analysis. Furthermore, all of this effort may be futile: when the number of clients is large, it becomes likely that *some* client is unreachable, such that the server must wait for that client's lease to expire before the write can proceed.

This poor behavior for writes is avoided if the server does not seek approval for writes, but instead simply waits for the leases to expire. Since all that the server then needs is the latest time of expiration, it does not even need to record the identities of the leaseholders, lowering the overhead of its bookkeeping. The cost of doing so is increased latency for the infrequent writes.

If the server gives the same expiration time to all of its clients, though, it runs the risk of synchronizing them so that they all the server runs the risk of synchronizing the clients, so that they all request extension together—and the implosion is not eliminated, but instead shifted to the extension traffic. When multicast is available, the server can avoid most requests for extensions by periodically multicasting an extension to the group of interested clients; a single multicast message per term then replaces a request and response for each active client. By multicasting before the term expires, the server can keep clients' leases from expiring, so that the contribution of coherence to response time

is minimized. The only requests for extension that the server receives are those for newly started clients and for clients that miss an occasional (datagram) multicast.

Without multicast, the server can still manage extension traffic, but not quite so simply. For the server to send unsolicited extensions, it must keep a record of the leaseholders, iterating through the list to send extensions. It can spread the extensions out in time, so that there are not bursts. Handling client requests for extensions requires looking up in the list, which either is expensive or requires a more sophisticated data structure than is suitable for small numbers of leaseholders. The server's other option is to not record the leaseholders, as with multicast, in which case it advances the latest-expiration time whenever an extension is requested.

There is still the potential for congestion when a write does occur. By continuing to send extensions, though, with the modified items explicitly excepted, the only extra traffic generated is for read accesses in the interval between the sending of the extension message and the expiration. This traffic includes queries to determine whether the data has changed, up until the write is performed, and cache misses to fetch the modified data thereafter. The latter traffic is unavoidable in any case.

Installed files are easily identified, residing in a few system directories, so that they can be specified as part of the file server's configuration to take advantage of these different policies. The result is that the cost of maintaining coherence on installed files becomes very low, with the only penalty being the increased delay incurred by the infrequent installations. When other groups of files that can be identified, knowledge of their access pattern can also be exploited. For example, when data is updated only periodically, leases can be granted to expire at the time of the next update, so that approvals need not be sought.

Adaptive policies

Some of the policy options can also be selected dynamically in response to access patterns. In fact, two simple cases of adaptation were already considered in the analysis of performance. By requesting extensions only when data is read, rather than periodically, the set of leases automatically represents the recent pattern of access, reducing the level of false sharing. By seeking approval only when the time to obtain it is less than the time remaining in a lease's term, the server automatically seeks approval less often as the level of sharing increases.

More explicit adaptations are also possible. The server can switch automatically to

treating a file as installed when the degree of sharing exceeds a configured threshold. Doing so allows the server to reduce the amount of state it keeps locally, since it does not have to keep the leaseholders' identities, in addition to eliminating traffic for approvals. The server can easily switch back to normal handling by ceasing to send unsolicited extensions and resuming keeping track of the clients that request extensions. Once the last unsolicited extension has expired, handling can revert to normal, or, if the number of clients requesting extension exceeds the threshold, the server can continue to treat the files as installed.

Another possible adaptation is for the server to dynamically change the term of which leases are granted. The only durations to consider, though, are zero and the maximum allowed, because any term in between them produces both higher traffic and increased delay. The switch to a zero term should be made when either α falls below one or the minimum for t_C exceeds the maximum allowed term. To estimate these values the server must have the actual rate of reads, which is known only to the caches; to allow the server to estimate the rate of reads, each cache would have to include in its request for extension the number of reads it has handled since the last request. The access patterns measured in V suggest that such switches would be extremely rare, since a very high degree of sharing is required to bring α below one; the need for switching is even smaller if installed files are configured for the special handling described above. For V or a similar system, then, the benefit from changing the term is probably too small to be worth supporting.

2.4.2 Other applications

The problem of coherence arises in contexts other than file caching, including multiprocessor memory systems (*e.g.*, [65]) and distributed shared memories (*e.g.*, [42]). What these applications have in common with file caching is the need to support a memory system that has multiple access paths and potentially multiple copies of data items.

These other applications have not traditionally considered partial failures; however, as larger multiprocessors are built and their interconnects more resemble networks, there is a growing need to address the potential for component failures. Leasing could prove useful in providing coherence in these contexts. The access patterns and time constraints may be quite different from those we have considered for file service; the trade-offs between delay and traffic would also differ. Performance could be in a region of the curves that we have not explored, where the time to obtain approval and the portion of the term lost to clock error and communication delays are a significant fraction of the term. The

trade-offs among different policies for extension and approval would need to be explored for these parameters, and possibly with a different distribution of operation arrivals.

2.4.3 Write-back caches

Some applications for coherence either do not require recoverability or provide it by other means. For example, a distributed virtual memory has no sense of persistence,[15] and in a transaction-processing system recoverability is enforced at a larger grain than individual operations. For such applications, leasing can be extended to support write-back caches, though it is then not possible to maintain coherence across all failures.

Write-back caches require a second kind of lease: a *write lease* authorizes its holder to read and write a data item during its term, provided it writes back any dirty items before the lease expires or is relinquished. Unlike the previously described read leases, though, a write lease is exclusive: all other leases over an item must be relinquished or expire before a write lease can be granted. Conversely, a write lease must be relinquished before a read lease—even with zero term—over the same item can be granted.

Because the system is not recoverable, coherence can not be assured for many failures. A client crash may lose newly written data; the only way to restore coherence in such a case is to undo all actions causally dependent on the lost writes, which is not generally possible.[16] Either a communication failure or a server crash can prevent a cache from writing back data before its write lease expires. Either of these failures forces the cache and the server to compromise. If the cache keeps the dirty data, coherence is violated, since the server may grant conflicting leases, but discarding the data is also undesirable. The server (after recovering, if it crashed) is faced with a choice of waiting for the cache to write back the data, thereby preserving coherence but losing availability of the covered data, or allowing further progress by granting new leases and leaving the cache with inconsistent data. From a practical standpoint, the best that can be done is to give the cache a chance to write back before other leases are granted; doing so reduces the chance of lost writes, but does not eliminate it.

A few other practical suggestions should be noted. When a write lease is about to expire but its holder has not finished writing back the data, the server should extend the lease as long as it is making progress. Also, in order to avoid conflicting claims from clients after a server crash, the server should record write leases on nonvolatile storage,

[15] Though some do support checkpointing.

[16] The exception is transaction processing systems, which are considered in Section 4.2.

so that it can honor any that have not expired when it recovers, and so that it can at least attempt to avoid data loss from those that have expired. This requirement raises the cost of supporting leases over that for write-through caches, where only an upper bound on the terms was required.

2.5 Summary

Coherent access is essential in order for file-service caching to be easily usable. Real distributed systems experience partial failures; so a practical solution to the problem of ensuring coherence must function correctly in spite of such failures. Leasing guarantees cache coherence even in the presence of crashes and lost messages, and it does so without reducing the availability of file service below that of the file service without caches. Because it makes explicit use of time, leasing does depend on well-behaved, though not perfect, local clocks.

Analytical estimates can be used to predict how the choice of lease term affects performance. How the long the term can be is restricted by the fact that the term is also the maximum delay that coherence can add to an operation when there is a failure. Generally, the best performance is given by either a term of zero or of the maximum acceptable length. A zero term is favored when writes to shared data dominate reads and when the rate of access is low. Otherwise, both server load and response time decrease as the length of the term increases.

For file service, a fairly short term yields excellent performance along with acceptable degradation in the event of a failure. For the access patterns measured in the V-system, a term of around ten seconds greatly reduces both server traffic and per-operation delay, and the additional gain from even a much longer term is very small. These results still hold when processor speed or network latency is increased.

Finally, leasing is flexible. It supports other policy choices in addition to selection of the term, and those options can adapt it for different access patterns and different trade-offs in performance. Furthermore, the mechanism is easily extended to support write-back caches, and it should be useful in other applications as well as in distributed file service.

Chapter 3

A File Cache for the V-System

This chapter describes a prototype file-service cache for the V distributed operating system. The design is evaluated in terms file-server traffic, based on analysis and simulation using traces collected with an instrumented cache. A very simple cache significantly reduces read traffic; the prototype's extensions to this basic cache yield further reductions in traffic of as much as 60% without compromising the robustness of the file service.

The first section describes the context for the prototype, and Section 3.2 introduces the basic cache design and the measurements that are the basis for evaluating performance. Each of the next three sections describes an extension to the cache design and evaluates its contribution to improved performance; those extensions are additional support for temporary files, for caching of descriptor information, and for maintaining coherence using leasing. Section 3.6 considers additional issues of the design as a whole, Section 3.7 discusses how the results generalize to systems other than V.

3.1 Background

The V-system is a distributed operating system based on the client-server model, with access to servers via location-transparent interprocess communication (message passing) [9, 14]. The V kernel provides a minimal set of services, with most operating-system services provided by user-level server programs. Where possible, services share common protocols, the most pervasive of which are the naming protocol [16, 46] and the I/O protocol [13]. The naming protocol defines a single global name space for all systems services, and application programs communicate directly with servers to resolve names and to invoke operations, in contrast with the *clerk* model of many systems, in which

each host has a local instance of each service, and an application program communicates only with that local representative.

Adding caching to V's file service involves two existing services, the file service and the kernel's memory service. This section describes the aspects of these services that are relevant to file caching.[1]

3.1.1 File service

File service in V is defined by a common interface that is supported by a variety of servers, such as a gateway providing access to the file system of a Unix host as well as the native V file servers. Other services can also conform to this interface, such as user-information, news, version or configuration management, and database services. To the extent that these services conform to the file-service interface, it should be possible to cache data from them using the same mechanism as for more traditional file service.

The interface for file service has three components: naming, I/O, and descriptor access.

Naming. The naming protocol provides a single tree of names seen by all programs executing on all hosts throughout the system. Names are interpreted by the servers implementing the named objects; clients cache hints as to the servers implementing different subtrees of the name space so that an operation on a named object can be efficiently invoked. By convention, a file server executing on host *hostname* implements the subtree of names rooted at `/storage/`*hostname*.[2]

Because names are interpreted by the servers, not the clients, a name can be viewed as a query against the server or servers that implement it. One use of queries is to support *generic* names. Instead of having a single binding, a generic name is bound to a set of objects; each operation using the name, however, acts on only one member of the set. For example, the root of the standard subtree of publicly available files is the generic name `/storage/any`, and an application uses that prefix to reference public files. There can be more than one server providing a copy of the public tree under that name, but each request is handled by only one server. Similarly, most file servers support the generic name `/storage/local`; an application can use this name to find local file storage.

[1] In the interest of clarity and simplicity, the descriptions here differ from the existing implementations in a few details that do not affect the nature of the results.

[2] For historical reasons, the current implementation uses '[' both for the root character and for the prefix for defined names. The descriptions here use '/' for the root and '%' for defined names.

The client naming library also supports *defined* names that are inherited by child programs. Defined names are used to designate specific functions and as convenient nicknames. For example, `%sys` is used as the root of the standard subtree of public files, `%tmp` as a suitable place for temporary files, and `%home` as the user's home directory. Default bindings for the standard names are provided as part of initializing a workstation, within standard library routines, or as a side-effect of logging in. These names can be rebound and others bound by the users. Most names are looked up relative to some defined name or the current directory, which is itself usually reached relative to some defined name.

I/O. The I/O protocol provides block-level access to objects specified by low-level identifiers called *handles*. A handle is usually obtained by performing an open operation with the name of the desired object; the handle returned identifies the server implementing the object. The open operation also returns a set of attributes of the object, such as its block size, its length, and whether it supports random or only sequential access. The I/O objects provided by file service correspond to open files, and they share the properties of allowing random access and behaving like storage, such that reading a block returns the previously written value. Other I/O objects, such as a network interface, do not share these properties.

An open file in V is treated as a transaction. Opening a file for reading provides an immutable snapshot of its contents, and when an open file is written, the modified blocks become visible to subsequent opens only after the open file is committed, which normally occurs when it is closed. This property implies that an application program or user never has to deal with a partially written file, even if a failure occurs while it is being written.

Descriptors and directories. For each name in the name space, there can be a *descriptor*, which is a tagged record of attributes corresponding to that name. For traditional file service, a descriptor includes the final component of the name, permission and ownership information, and other attributes such as the file's size and last-modified time. The attributes can differ for other servers.

Directories are read using the I/O protocol, with a block corresponding to the descriptor for each name in the directory. In general, though, a client cannot resolve names by reading and interpreting directories. First, not all directories are readable: access controls can prohibit reading a directory, even when the user is permitted to look up names in it. Also, because a name is interpreted by the server, a directory need not be

enumerable at all: the server might compute a result using the tail of the name as an argument. Finally, directories that can be read do not necessarily contain any low-level identifiers that a client could use to for resolving the names in it.

3.1.2 Memory service

The virtual memory provided by the V kernel is all based on mapped I/O. An address space consists of a set of bindings, each from a region of pages into a range of blocks in an I/O object. The pages are specified by address and length, and the blocks by a handle and block offset. Page-in and page-out are done as reads and writes using the I/O protocol. In order to be mappable, an I/O object needs to have the same properties as an open file, plus it must have fixed-size blocks of a size compatible with the page size (*i.e.*, one size is an integral multiple of the other).

Two other features of the memory service complete the base for file caching. First, an address space can be created and its bindings manipulated without any processes executing in it; operations identify the address space using an I/O handle. Second, the memory server can create handles for an address space by which it can be read or written using the I/O protocol. For I/O using these handles, each page is a block, and handles can be created with restricted (*e.g.*, read-only) access. Reading and writing address spaces has long been used for non-paged program loading and for debugging. An address space accessed in this manner can be larger than the processor address space; the I/O protocol presently limits an object to 2^{32} blocks (pages).

These features combine to enable the caching of file blocks using the memory service. A program can open a file, create an address space and bind the open file to it in its entirety, and then create a handle with which to read or write the address space. For I/O, the handle for the address space is equivalent to the handle for the open file; the only differences are that it can be more restrictive in the access it allows and that the block size corresponds to the page size, which can differ from that of the open file. Blocks are faulted in as necessary to satisfy read requests, and a commit causes any modified pages to be written to the bound open file before the commit request is forwarded to the file server.

When the address space is created, an alternate file handle can be specified to field non-I/O operations against the address space; the memory server substitutes the alternate handle in the request and forwards the operation to the server it indicates for processing. Close operations, including commits, are also forwarded to the alternate

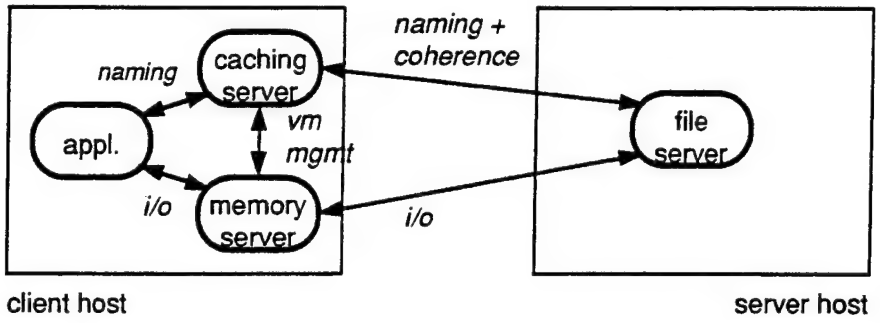


Figure 3.1: Servers and protocols involved in file caching.

handle after any required flushing is completed. The default for the alternate handle is the bound file handle, so that all operations behave as if they were requested directly of the file server.

Caching in this manner has a few limitations: it is inconvenient, as each program has to explicitly set up caching for each file it uses, and it is inefficient, since no caching occurs across program invocations.

3.2 The caching server

A process-level *caching server* on each workstation provides the additional functionality needed to make caching of open files practical: naming, re-use, management of the cache, and coherence. The caching server fields naming requests, and when a file is opened it arranges for block access via the local memory server in the manner described above. The caching server operates at the level of naming operations and whole files, while the memory server handles all of the block-level access to open files. Figure 3.1 shows an application and the servers involved in caching a file, along with the protocols and pattern of communication.

This section describes the caching server at a basic level of functionality that serves as a starting point for the improvements detailed in subsequent section; it also serves as a baseline against which the performance of those improvements is compared. The basic cache handles only file contents; it makes no effort to cache information about files, such as that contained in their descriptors, nor does it cache directories. The basic cache also uses on-use checks to ensure coherence, and it treats all cached files uniformly.

3.2.1 Description

In order to cache data, the caching server must intercept the naming requests destined for the file servers. The prototype has not addressed the problem of transparent naming, as doing so would require major extensions or changes to the way names are resolved in V. Instead, the caching server on each workstation implements the portion of the naming tree with the prefix `/cache/`*hostname*`/read-write`, and it strips this prefix from the name in a request to determine the real name of the object on which to operate. An application routes a naming request through the cache by adding the prefix to the name of the operand. In actual practice the added prefix is not obtrusive: it can be hidden in the binding of defined names such as `%sys` or `%home`, so that the user does not normally need to be aware of it. Also, each caching server implements the generic name `/cache/local` as an alias for `/cache/`*hostname*, so that the defined names can be bound without reference to a specific host.

Data structures. The caching server's central data structure is a tree of the names for which it has cached data. Each node in the tree contains the node's name, links to its parent and children, and a list of any versions of the corresponding file that are currently cached.[3] The descriptor for each version includes distinct handles by which it identified to the different servers, the identity of the user who opened it, a version tag, and flags indicating its mode and status. The contents of a version descriptor are summarized in Table 3.1.

Opening files. When it receives an open request, the caching server looks up the name in its tree to see if the file is already cached. If the name is not found, the caching server makes the open request itself after stripping its prefix from the name, so that the request will be fielded by the correct file server. The user identity and mode of the original request are unchanged, so that the file server can check permission just as it would for a request directly from an application. If this open request fails (*i.e.*, the file does not exist or the user does not have permission to open it), the caching server simply returns the indication of failure to the requesting application.

If the open is successful, the caching server attempts to set up caching from the handle returned by the file server: it asks the memory server to create a new address space and to bind the open file handle to it in its entirety. If the memory server indicates

[3] In the interest of brevity, we omit from our description some of the fields in naming nodes and version descriptors, used for purposes such as managing the cache's contents.

Field	Meaning
name	Points to the naming node for this file.
FShandle	Handle returned by the file server when the file was opened.
VMhandle	Handle for the address space into which the open file is bound; returned by the memory server when the address space is created.
CShandle	Handle by which the memory server identifies the open file to the caching server; provided by the caching server.
mode	Indicates whether the file as opened at the file server is writable.
user	The user who originally opened the file.
idle	Indicates whether any application presently has this version open.
status	Indicates whether this version is *past*, *current*, or *future*.
tag	Version identifier used in coherence checks.

Table 3.1: Contents of a version descriptor.

that the open file is not cacheable, however, the caching server returns the handle to the application so that all further interaction for the open file is directly between the application and the file server. Access to non-cacheable files is therefore supported, with the only cost being a small amount of additional overhead when the file is first opened.

When the open file is successfully bound into its own address space, the caching server adds the name to its in-memory tree and allocates a descriptor for the version. It sets the name, FShandle and VMhandle, and it allocates a new value for the CShandle, and it fills in the mode and user from the original request. If the file is opened for writing, its status is *future*, since it represents a not-yet-committed version of the file; the status for a read-only open is *current*. For a read-only open, the server also requests from the file server a version tag corresponding to the handle, which it stores in the naming node; the tag is 128 bits that the server can interpret later to determine whether the version is current.[4] Finally, the caching server flags the version as busy, requests from the memory server another handle for the address space, and returns that handle to the application.

[4]As presently implemented, the version tag is obtained by reading the file descriptor from the server and extracting suitable fields; *e.g.*, for the Unix-based file server these are the inode number and modified time for the inode.

I/O operations. The application uses its handle for the address space for all of its operations on the open file. This handle points to the memory server, and reads and writes are handled without involving the caching server.

The caching server specifies its CShandle as the alternate handle when it creates the address space into which the open file is bound. Because the memory server forwards all requests other than reads and writes to the alternate handle, these requests are intercepted by the caching server. When the caching server receives one of these requests, it maps from the CShandle to the corresponding FShandle and makes the corresponding request of the file server. The caching server then returns the result to the application; if the result includes a name (*e.g.*, mapping from an open file handle to its name), the caching server first adds its prefix to the name.

The caching server intercepts closes and commits by the same mechanism. Before forwarding a commit, the memory server writes out to the file server any modified pages; the caching server therefore completes the commit by sending the request to the FShandle, setting a flag in the request to indicate that the file should remain open. After the commit, the caching server requests a new version tag from the file server and stores it in the descriptor. It then invalidates any version marked as current: its status is changed to *past*, and if it is idle it is discarded by releasing the corresponding address space and closing the corresponding handle at the file server. If the commit is part of a close operation, the version's status is changed to current, and it is flagged as idle.

When the application closes a file that it has had open for reading, the caching server marks the version as idle if the flags supplied by the memory server indicate that no other application has it open. When a past version is idled, it is discarded.

Reopening cached files. When an open is requested for which the name exists in the tree, the caching server searches the list of versions for one that can be used to satisfy the request. If the requested mode is read-only, then any current version is usable. When a file is being opened for writing, then in addition to being current, the cached version must be both writable and idle as well. If no usable version is found, the request is treated as a cache miss, otherwise the caching server queries the server with the name, mode, and version tag to determine that (a) the version tag is not out-of-date, and (b) the requesting user has permission for the open. If this coherence check returns `STALE_DATA`, the current version is invalidated and processing continues as for a cache miss; any other error result

(including NO_PERMISSION) is returned to the application.[5] If the check returns OK, the caching server obtains from the memory server another handle for the corresponding address space and returns it to the application. The handle returned to the application has only the access rights requested in the open mode: a read-only open receives only read rights, even if the cached open file is writable.

Other naming operations. For any other naming request it receives, the caching server strips its prefix from the name and issues the modified request. The results are returned to the application, after modifying any name in the results to include the caching server's prefix.

This basic cache provides a starting point; its level of functionality is quite similar to the first version of the Andrew file system [58] as well as to that of Sprite file [50] without delayed write. (A more detailed comparison with these and other systems appears in Chapter 5.)

The basic cache introduces no problems with respect to failures. Data is always written through to the file server before it is committed; so a client crash cannot cause data loss. Likewise, applications on a client host can read and write data whenever it can communicate with the file server, which is the same availability as in the absence of caching.

The caching server requires no special status with the file servers; it uses the same interface as other clients of the file service, with the only addition being the on-open coherence and permission check. In particular, a file server does not have to trust the caching servers to enforce protections, since the file server still enforces protections on each open that it processes. The users of a caching server must trust it to act on their behalf and to enforce protections between multiple users on a single host.

3.2.2 Traffic measurements

Throughout this chapter, the impact of design choices is evaluated in terms of the traffic handled by the file server. The counts of operations are based on traces collected using

[5] Presently, the caching server requires that the version have the same user for read-only opens as well, and the coherence check is performed by reading the descriptor and comparing the stored fields. The modified time for the attributes (instead of the contents) is used, so that if the permissions have changed the check returns STALE_DATA, forcing a cache miss, and the normal permission check is performed as part of handling the miss. In a similar manner, an extra cache miss occurs when a second user opens a file already in the cache. In the V installation at Stanford, neither of these cases occurs frequently enough to produce a perceptible degradation in performance.

an instrumented version of the caching server that records all file-service operations that it processes or invokes. The trace entry for each operation includes the operation, its outcome, the full names of its operands, and a timestamp. Individual reads and writes are not traced, since they are handled by the memory server instead of the caching server; however, counts of blocks read from and written to the cache are included in the record for each close. Reads and writes handled by the file server are not captured, but could be estimated. A few other operations on open files are not traced, but their use can be reliably determined for the traces presented here. More detailed information about the trace data and estimates appears in Appendix A.

Measurements from three traces are presented here. Each captures an activity on a single workstation. The activities traced are:

`fsbuild` Set up working directories for a private version of one of the V file servers, then compile it, edit a header file, and recompile.

`afsbench` The Andrew file system benchmark [35], modified to run under V: copy a directory subtree, scan all of the directories and files in the new copy, and compile sources in it.

`latex` Format a conference paper by running the LaTeX program twice.

Two of these traces, `fsbuild` and `latex`, capture normal activities under V, while the other, `afsbench`, is of a synthetic benchmark. All three represent intense bursts of activity corresponding to the peak, rather than average, rate for an interactive user. While most of a user's time is spent in less intense activity, the greater part of the load on file service is generated in bursts of activity like those traced.

All three traces described here were collected on a MicroVAX II workstation with sixteen megabytes of memory, accessing files from a variety of remote file servers, as is common in V. Apart from the traced activity, only normal system functions were active on the traced workstation, and they generated no file operations. All traces were collected starting with a cold cache, and none required any replacement of data in the cache.

For conciseness, all of the measurements are presented with the operations are grouped into the following categories:

read One-kilobyte blocks read.

write One-kilobyte blocks written.

commit Operations that commit a visible change, such as a close after writing, or file creation or removal.

naming read Other operations that read naming or descriptor information, such as opening an existing file or reading a descriptor.

misc. Miscellaneous operations, such as non-committing closes or file truncation.

coherence Operations required to ensure cache coherence.

Basic measurements

The baseline for comparison is the level of traffic with no cache, which is summarized in Table 3.2. The same data is shown in Figure 3.2 in terms of the *traffic ratio*, which is obtained by dividing each count by the total traffic for the trace. Reads dominate the traffic for all three traces.

A second initial data point is the performance of the basic cache described in the preceding subsection. Table 3.3 and Figure 3.3 show the traffic for such a cache. Traffic is significantly reduced, mostly in blocks read. The savings in blocks read comes from two sources: files that are read more than once, and files that are written and then read. The gain for `latex` is more modest than for the other traces: because `latex` is much shorter, its read traffic is dominated by the effects of starting with an empty cache.

Read traffic cannot be reduced any further: once a block was present in the cache, it was never read from the server again. In all three cases the cache was large enough to hold all of the data used over the entire trace. For longer-term operation, some cache replacement would occur, but the additional reads required when a replaced block is later read would be offset by a much smaller contribution from cold-start effects. Any further reduction in traffic must therefore be found elsewhere. Three components of traffic stand out as candidates: writes, naming reads, and coherence.

Traffic by file class

Files are not all alike. Breaking down the traffic measurements by file classes reveals very different patterns of access. The classes considered here are:

Temporary files. Temporary files are commonly used to hold intermediate results, either as an extension of a program's address space or for communication between programs (such as between passes of a compiler). In many systems, including Unix

	fsbuild	afsbench	latex
read	73315	25025	1846
write	21237	3808	124
commit	1263	476	8
naming read	11297	3426	182
misc.	5269	952	90
total	112381	33687	2250
duration	7521 sec	1653 sec	263 sec.

Table 3.2: Traffic without caching.

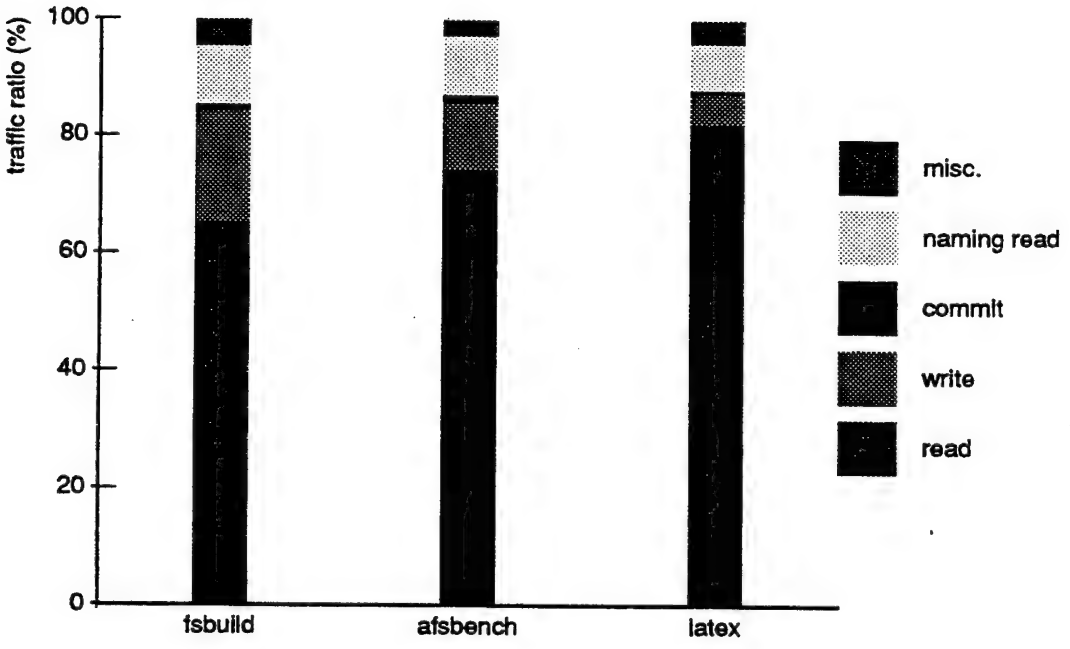


Figure 3.2: Traffic without caching.

	fsbuild	afsbench	latex
read	3284	1495	872
write	21237	3808	124
commit	1263	476	8
naming read	6147	2567	104
misc.	510	267	16
coherence	5150	859	78
total	37591	9472	1202
traffic ratio	33.5%	28.1%	53.4%
read traffic ratio	4.5%	6.0%	47.2%
max. cache size	9 MB	5 MB	3 MB

Table 3.3: Traffic with the basic cache.

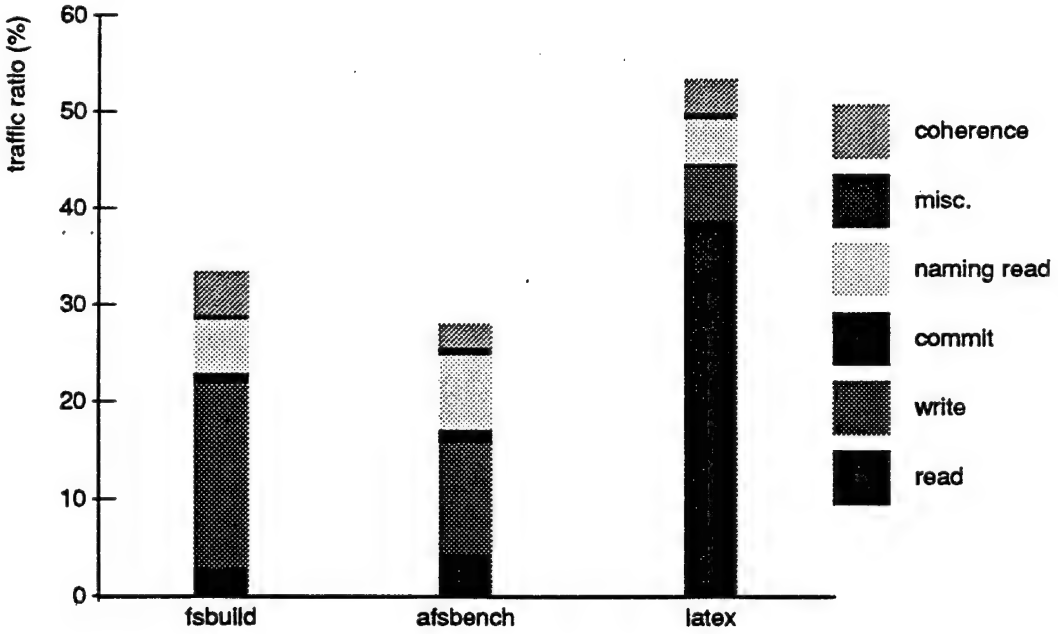


Figure 3.3: Traffic with the basic cache.

and V, one or more directories are provided, by convention, for temporary files; others, such as Tops-20, allow a file to be tagged as temporary when it is created. In either case, temporary files are easily identifiable.[6]

Installed files. In any system, there is a body of files that is not updated in normal operation, but only when a new version is installed. Examples are the executable form of commands, object code libraries, and on-line documentation; public source code is often treated in a similar fashion. These files are widely shared among users, and some of them are very heavily accessed; that shared access, however, is predominantly read-only, since installations are relatively infrequent. Installed files reside in a fairly small set of public directories or subtrees, and for administrative reasons (*e.g.*, prevention and detection of unauthorized modification) it is usually desirable to segregate installed files from other public files that are regularly updated.

Other files. The remaining files are grouped together. Note that the traces considered here include only user access to files, so that the measurements do not include any access to files used in either the interface or implementation of system services.

These classes represent two different types of distinctions. Installed patterns are distinguished by the pattern of access to them; they are semantically the same as other files. The difference in access pattern suggests that separate tuning for them could improve overall performance, but with no changes in functionality.

Temporary files differ in kind, to a degree that it is inaccurate to classify them as files: one of the primary purposes of files is persistence, a property that temporaries do not share. Including access to temporaries in overall file access patterns is therefore misleading. This observation does not reduce the need for temporary data storage, nor does it suggest that the interface to such storage should differ from that for files—the common interface allows the substitution of one for the other. But the needs of temporaries might be better met in ways other than reliable storage on disk.

Table 3.4 and Figure 3.4 break down by file class the traffic without caching. Within any single trace, the traffic patterns differ greatly from class to class. For each class, however, the makeup of its traffic is similar across all of the traces. Much of the difference between traces is accounted for by different amounts of traffic for each file class. The `latex` trace shows the largest variation: it uses no temporary files at all.

[6]There are Unix programs that create files in these directories (`/tmp`) that do require persistent storage, such as the journal files generated by some editors. None of these programs exist in V; they indicate a need for a different class of persistent "scratch" space, located in a different directory.

	fsbuild			afsbench			latex	
	tmp	inst	other	tmp	inst	other	inst	other
read	13872	42265	17178	2625	17858	4542	1732	114
write	11596	0	9641	1773	0	2035	0	124
commit	1020	0	243	222	0	254	0	8
naming read	338	3970	6989	70	957	2399	72	110
misc.	338	3030	1901	70	480	402	70	20
total	27164	49265	35952	4760	19295	9632	1874	376
share of total	24.2%	43.8%	32.0%	14.1%	57.3%	28.6%	83.3%	16.7%

Table 3.4: Traffic without caching, by file class.

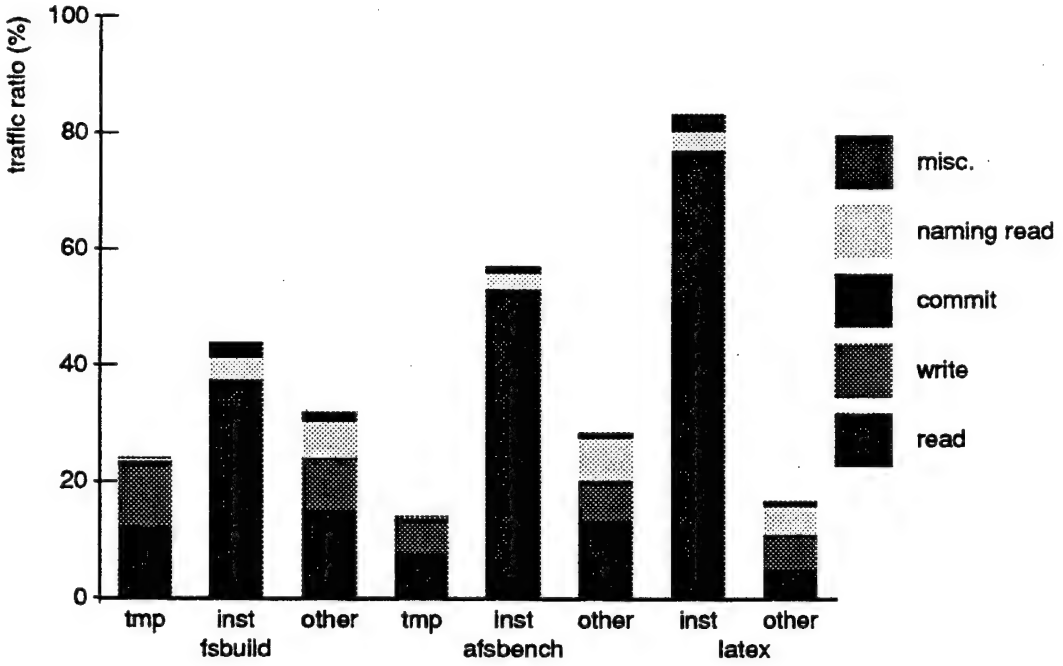


Figure 3.4: Traffic without caching, by file class.

	fsbuild			afsbench			latex	
	tmp	inst	other	tmp	inst	other	inst	other
read	0	2763	521	0	1093	402	820	52
write	11596	0	9641	1773	0	2035	0	124
commit	1020	0	243	222	0	254	0	8
naming read	0	1034	5113	0	501	2066	12	92
misc.	338	94	78	70	24	173	10	6
coherence	338	2936	1876	70	456	333	60	18
total	13292	6827	17472	2135	2074	5263	902	300
traffic ratio	48.9%	13.9%	48.6%	44.9%	10.7%	54.6%	48.1%	79.8%
share of total	35.4%	18.2%	46.5%	22.5%	21.9%	55.6%	75.0%	25.0%

Table 3.5: Traffic with the basic cache, by file class.

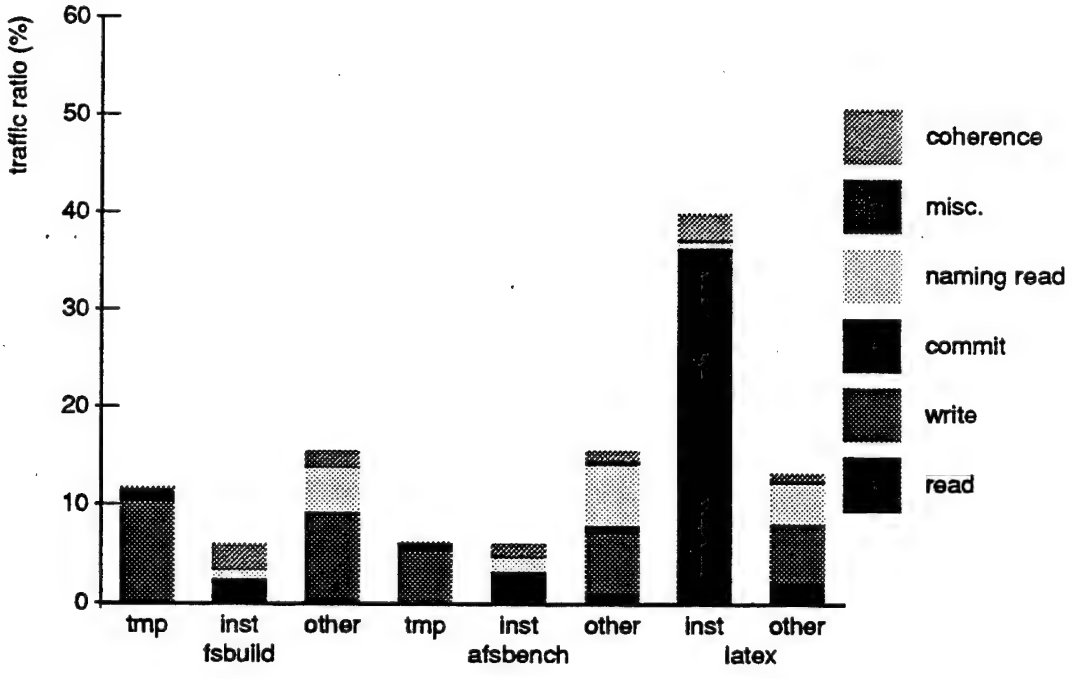


Figure 3.5: Traffic with the basic cache, by file class.

That `latex` does not use temporary files reflects the more recent origin of the programs it executes. Much use of temporaries in Unix software dates from a time when address spaces and main memories were both more limited in size, so that it was common for either code or intermediate results to exceed the available space. More modern programs are more likely to retain intermediate results within their address space, and they are less likely to require that code be split into multiple programs. It is therefore reasonable to anticipate some decline in the use of temporary files as more modern software, such as the TEX formatter, displaces older programs, such as the Portable C Compiler used in both `fsbuild` and `afsbench`. Their use is not expected to vanish altogether, however, as there are reasons other than size constraints for separating a program into multiple address spaces So the need will continue for efficient temporary storage that is globally nameable and is sharable, and that is interchangeable with persistent files.

The same general observations hold for traffic with the basic cache, as Table 3.5 and Figure 3.5 show; in fact, the differences between classes are more pronounced with caching. Temporary files produce little traffic apart from writes and commits: reads are eliminated since all reads are of files just written. Installed files benefit from repeated reading, though they also suffer the most from start-up misses in the `latex` trace; the reuse of installed files produces a significant amount of coherence traffic. For the other files, both writes and naming reads account for significant portions of the traffic; reads are reduced both due to reuse and, as for temporary files, due to reading back newly written data.

Sharing

The traces contain no data on sharing, since each is of a single user's activity on a single workstation. We can make some general observations based on published measurements of sharing in timesharing systems [11, 26, 37, 49, 65]. (A fuller discussion of these and other measurement studies appears in Section 5.2.)

As previously noted, installed files are shared widely, and the most heavily accessed of them are shared by all users. Temporary files are almost never shared among users; most are accessed by only one or two processes. Sharing of other files is less uniform, but most of the observed sharing is of files used in the interface to some service—such files do not exist in V. Of the remaining files, most are accessed by only one user at a time. A small fraction are shared among a small number of users, but the shared files are less frequently written than those that are not shared. The decrease in frequency of writes

with increased sharing is important, because it implies that more expensive writes, in terms of number of approvals required, occur less often.

In most existing systems, a user is able to easily employ only one workstation at a time, so that sharing among users does describe sharing among hosts. But a few systems, including V, allow a user to invoke commands transparently on additional hosts [64]. Using this facility creates additional sharing between workstations. But the scheduling of execution on other hosts clearly needs to consider cached data, which the existing V mechanisms do not; measurements of sharing using the existing scheduling could not be considered representative.

A simple approach to scheduling that would yield some locality of reference is one in which a user employs a small set of hosts over an extended period, with hosts added to or removed from the set as needed. Reusing the same set of hosts gives some benefit from caching across multiple program executions, which is not likely if a new host is selected at random for each command. Under such a scheduling policy, there is much more sharing, including write-sharing, of other files, but each file is likely to be shared by only a small number of workstations.

3.3 Temporary data

Temporary files account for a significant fraction of the traffic in two of the traces. With the basic cache, most of the traffic for temporary files is the writes and commits required to make them persist reliably, even though persistence is not required for temporary data. The caching server eliminates almost all traffic for temporary files by providing additional support for temporary data, taking advantage of the fact that persistence is not required. The support required adds little to the complexity or size of the caching server. Special handling for temporaries yields performance benefits comparable to delayed write, a common approach to reducing write traffic, but without compromising the reliability of file data as delayed write does.

3.3.1 Caching server support for temporary data

In addition to the subtree of cached files, each caching server provides a directory /cache/*hostname*/tmp in which temporary files can be created, as an alternative to caching such files from a file server. Because the files are implemented locally, the overhead of creation, deletion, and checking coherence is avoided. These temporary files,

however, are still globally accessible and sharable, since they are within the shared name space and are accessed via the standard I/O protocol.

The temporary support uses the memory server for block-level I/O, just as for cached files. Because blocks are cached in virtual memory, there must be an open file to which they can be paged. The caching server maintains a pool of anonymous open files for this purpose, using one per temporary file created; they are recycled, however, requiring only a truncation to zero length to clear their contents before they are reused. Because few temporary files are in use at one time, a small pool suffices, with the caching server adding to it as needed. In addition, the caching server directs the memory server to *not* write back dirty pages on commit. Blocks are written back to the file server only as required for page replacement; with a memory of adequate size, no data from temporary files is written out. The only traffic required for temporaries, then, is the creation and deletion of files for the pool and the truncation each time a file in the pool is recycled.

Table 3.6 gives the traffic levels after adding temporary support to the basic cache; Figure 3.6 presents the same data in terms of traffic ratio. Traffic for temporaries is essentially eliminated. (The `latex` trace is not included, since it uses no temporary files.) Combined traffic for all classes is presented in Table 3.7 and Figure 3.7. For `fsbuild`, writes are reduced by 55% and committing operations by 80%; for afsbench the reductions are 47% and 45%. Total traffic is reduced 34% for `fsbuild` and 22% for afsbench.

The cost of support for temporary data is small. In the prototype, it adds four classes that are implemented in about 825 additional lines of C++, of 11,000 lines for the entire caching server (not including libraries). On a MicroVAX, the added code is less than eight kilobytes out of the caching server's total of 140 kilobytes, plus roughly four kilobytes of additional data at run-time.

3.3.2 Comparison with delayed write

We now compare our special support for temporary data with a more common approach to reducing write traffic, delayed write. In centralized Unix systems, a write system call returns as soon as the data has been copied to a system buffer, and it is written to disk sometime later [66]. Delaying the write to disk lowers the latency for the write operation, since the operation completes without waiting waiting for the disk, and it reduces the level of traffic at the disk, since a significant fraction of newly written blocks are rewritten or deleted within a short time. The performance improvement is potentially greater in a

	fsbuild			afsbench		
	tmp	inst	other	tmp	inst	other
read	0	2763	521	0	1093	402
write	0	0	9641	0	0	2035
commit	8	0	243	6	0	254
naming read	0	1034	5113	0	501	2066
misc.	340	94	78	74	24	173
coherence	0	2936	1876	0	456	333
total	348	6827	17472	80	2074	5263

Table 3.6: Traffic for cache with temporary support, by file class.

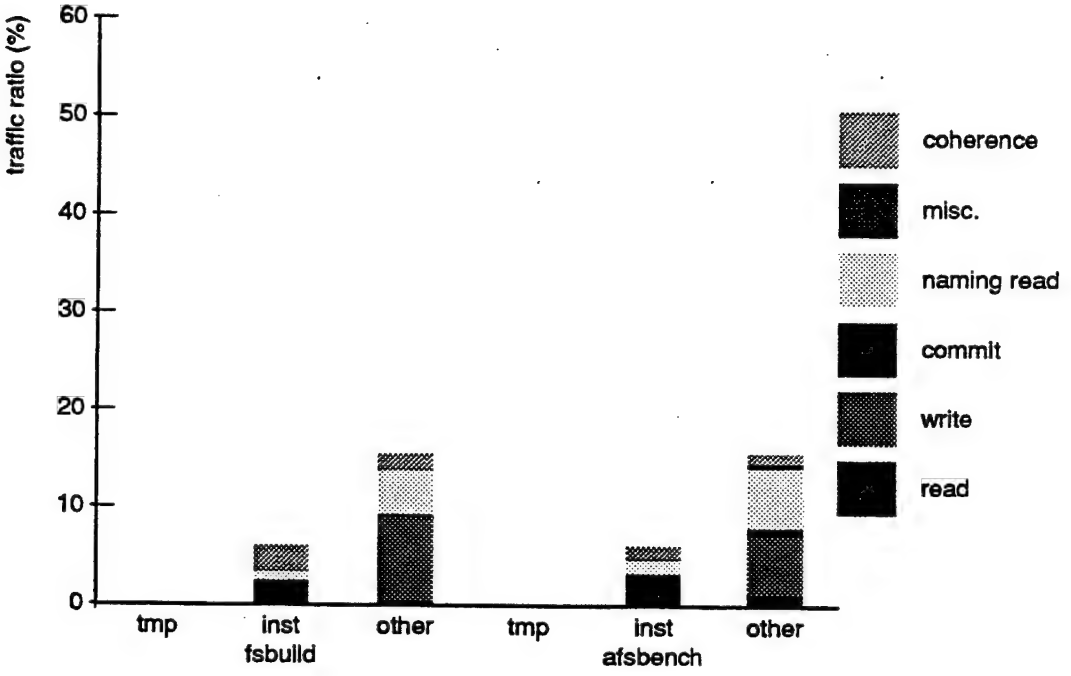


Figure 3.6: Traffic for cache with temporary support, by file class.

	fsbuild	afsbench
read	3284	1495
write	9641	2035
commit	251	260
naming read	6147	2567
misc.	512	271
coherence	4812	789
total	24647	7417
traffic ratio	21.9%	22.0%
relative to basic cache	65.6%	78.3%

Table 3.7: Traffic for cache with temporary support.

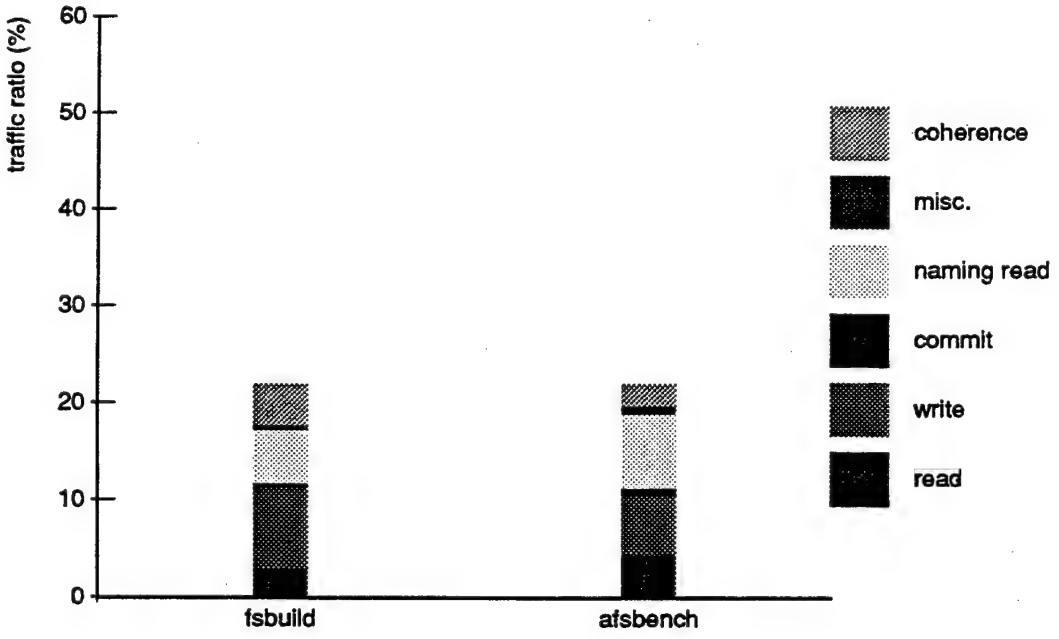


Figure 3.7: Traffic for cache with temporary support.

distributed system, where the disk write becomes a write to a remote, shared file server. Delayed write therefore plays a central role in several distributed file systems, including Sprite [50], Burrows' MFS [11], and Echo [45]. Our design compares favorably with delayed write in all three of reliability, complexity, and performance.

Reliability. The primary drawback of delayed write is that it places all newly written data at risk of loss when a crash occurs. To limit the amount of loss that can occur, the delay is bounded by having a system process periodically write out any dirty blocks; a typical interval is thirty seconds. When a failure occurs, then, some of the data written during the last thirty seconds can be lost. Section 1.2 points out that even if this risk were acceptable in a centralized system, the nature of the failures that occur in a distributed system make the risk unacceptable there: loss can result from the failure of any of client, server, or network, and the loss of data due to a partial failure can go undetected. In contrast, the only data placed at risk by the caching server's approach is that in temporary files.

Complexity. The principal argument advanced against special support for temporaries is that it adds unnecessary complexity. The implementation, however, can be quite simple, as the prototype demonstrates. And any added complexity for applications is hidden within standard library routines for creating temporary files.

Delayed write introduces complexity of its own. The cache coherence mechanism must be prepared for the latest version of a file block to exist somewhere other than the server. Coping with failures is much more complicated, since a recovering server must somehow find blocks that have not been written back from client caches, and a cache can find itself unable to write out committed data due to failure of either the server or communications. The complexity filters up to the application or user, which must either be prepared to cope with loss of data or take additional steps to ensure its persistence.

Performance. Delayed write has two main benefits for performance: reduced write traffic, from avoiding writes of short-lived data, and lowered response time, since writes do not wait for either remote communication or disk access.

For our traces, delayed write does not reduce traffic as much as does support for temporaries. The reduction in write traffic depends on the lifetime of newly written data, which for the traces is shown in Figure 3.8. For `latex`, no data lives for less than two minutes; in the other traces, about half survives for under thirty seconds, while the

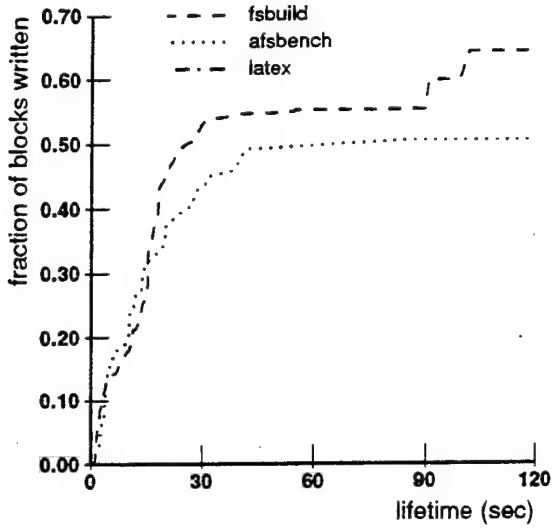


Figure 3.8: Lifetime of newly written data in all files (cumulative).

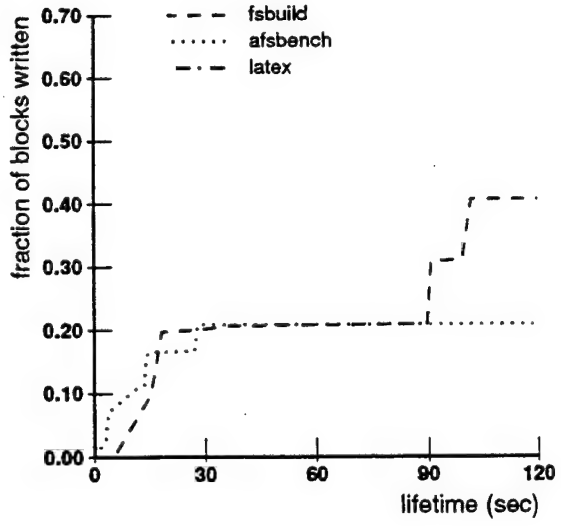


Figure 3.9: Lifetime of newly written data in non-temporary files (cumulative).

remainder survives much longer. Figure 3.9 shows lifetimes of blocks written to non-temporary files; there is little short-lived data in non-temporary files. Table 3.8 gives the traffic for a simulated cache using a thirty second delay of all writes; for both `fsbuild` and `afsbench` the write traffic is higher than with support for temporaries, because some temporary files survive for more than thirty seconds,

Figure 3.10 shows traffic for a thirty second delay in comparison with that for supporting temporary data directly. While write traffic is comparable, the support for temporaries also eliminates most of the operations to create, commit, and delete temporaries as well as all coherence checks. The higher commit and coherence traffic results in a thirty second delay producing a slightly higher level of total traffic.

The other performance benefit claimed for delayed write is that it lowers response time, since writes and closes are asynchronous. In V, however, block-level write operations are already asynchronous—the only requirement being that they complete before the commit. So the difference in response time for the two approaches depends on the number of synchronous operations each requires. For delayed write, file creation and deletion still require synchronous operations, while file commits are asynchronous; the support for temporaries eliminates creation and deletion for temporaries while still requiring synchronous commits of non-temporary files. Table 3.9 gives the number of synchronous committing operations for both approaches. Delayed write shows a higher number of these operations for all but the `latex` trace, and for that trace the number is quite small. While the relative delay for these operations depends on how the server implements them, we can conclude that any difference in response time for the two techniques can be expected to be quite small, and that support for temporaries is likely to outperform delayed write in many cases.

The traces are from execution on a fairly slow processor. With a much faster processor, file lifetimes should decrease, which would improve slightly the relative performance of delayed write. The lifetimes of temporary files are bounded by execution time; so a greater share of temporaries would benefit from delayed write. The survival of other files, however, normally depends on some human interaction, and so their lifetimes would shorten less. To the extent that the speed of human interaction increases, however, the risk from a crash also increases: more activity is subject to loss, with greater effort required to recover it, and, with reordering of delayed writes, the number of possible states in which a user or application might find a set of files increases greatly. Even on much faster processors, the cost of reliable file storage is expected to remain low.

	fsbuild	afsbench	latex
read	3284	1495	872
write	9984	2156	124
commit	1263	476	8
naming read	6147	2567	104
misc.	510	267	16
coherence	5150	859	78
total	26338	7820	1202
traffic ratio	23.4%	23.2%	53.4%
relative to tmp support	107%	105%	100%

Table 3.8: Traffic for cache with 30-second delayed write.

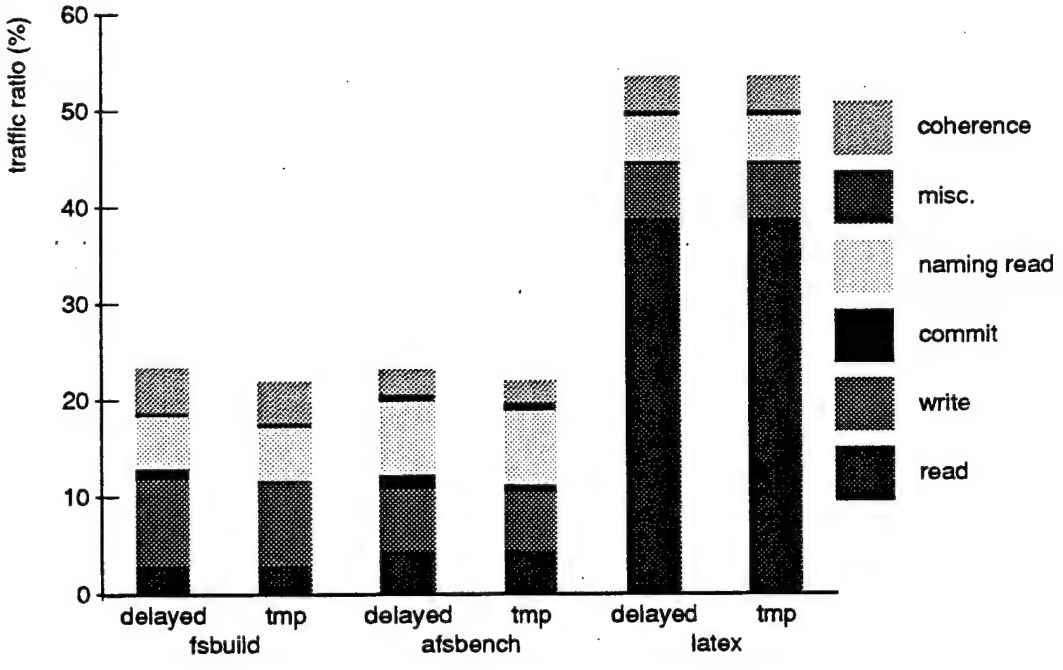


Figure 3.10: Traffic for 30-second delayed write vs. temporary support.

	fsbuild	afsbench	latex
delayed write			
file creation	412	176	0
file deletion	340	74	0
total	752	250	0
temporary support			
file creation	76	103	0
file deletion	0	0	0
file commit	125	104	8
total	201	207	8

Table 3.9: Synchronous commit traffic.

In summary, special support for temporary data yields a substantial reduction in traffic without sacrificing the reliability of the file service. Providing this support adds very little complexity to the caching server, and preserving the robustness of file service simplifies the writing of robust applications. This approach compares favorably in all respects with the more common approach of delaying writes, which reduces write traffic at the expense of reducing reliability for all files.

3.4 Caching descriptor information

The basic cache handles only the contents of files, but the file service also stores information about files in the form of their descriptors. With file contents cached, a large portion of the file-server traffic is for read-only access to descriptor information; in addition to the explicit naming reads, each on-open coherence check also reads this information to check the name and permissions. Table 3.7 shows that for the basic cache with temporary support, 15–45% of the remaining traffic falls in this category. Caching descriptor information therefore offers an opportunity to significantly reduce traffic.

There are two consumers of the information about files: application programs and the caching server itself. Applications read directories and descriptors; if the cache is to avoid performing any of the naming reads or coherence checks, then it needs this information to perform name lookups and permission checks.[7] A significant fraction of those name lookups fail: 18–48% of the names looked up in the traces are for files that do not exist.

[7]Even though the file server enforces permissions, the cache must also do so when permissions change after a file is cached and when the cache is shared by multiple users.

Field	Meaning
name	The name of this node.
parent	Pointer to its parent node.
children	List of child nodes.
versions	List of version decriptors for cached files.
descriptor	The name's descriptor.
volatileValid	Indicates whether the volatile attributes in the descriptor are valid.
stableValid	Indicates whether the stable attributes in the descriptor are valid.
childListValid	Indicates whether children is a complete and valid list of the children of this directory.
users	Table of users known to have permission to look up this node's name.

Table 3.10: Contents of a node in the naming tree.

This section describes and evaluates extensions to the basic cache to allow it to cache descriptor information as well as the contents of files. The nodes within the naming tree require additional fields in order to store this descriptor, which are listed in Table 3.10 along with the fields from the basic cache. In addition, a *permissionChanged* flag is added to the descriptor for a file version.

When the caching server reads the descriptor for a file (or directory), it caches it by copying it into the corresponding node of the in-memory naming tree and sets the both atrribute flags to valid. he caching server reduces the frequency with which it must read descriptors from the file server, however, by caching results of permission checks, by caching entire directories in response to failed name lookups, and by partitioning the information within a descriptor to invalidate as little as possible.

Caching permission information. A successful operation returns additional information implicitly along with its explicit result. In particular, a successful operation that includes a name lookup implicitly indicates that the user has permission to look up each component of the name. When a lookup operation at the file server succeeds, the caching server adds the requesting user to the table for each node along the path. Subsequent lookups by the same user can be permitted without having the actual descriptor data. Similarly, the descriptor for each cached file version includes the user who opened it and the mode in which it was opened, indicating that the same user has permission to

open it again in the same or a more restricted mode. Caching these results of the open request enable the caching server to handle a repeated open without having cached the descriptor for every component of the path.

When the caching server needs only additional permission information to perform an open, it can obtain that information by reading a single descriptor. For example, when a second user attempts to open for reading a file that is already in the cache, reading the file's descriptor by name provides implicitly an indication of the user's authorization to name the file along with the explicit protection information from which the cache can determine whether the user has permission to read it.

Handling failed name lookups. The cache handles failed name lookups by caching entire directories: if the name lookup fails in the cache's tree, then the named file does not exist. A directory is cached, however, only after it is read by an application or a name lookup within it fails. For example, when an application attempts to open for reading a file that is not in the cache, the caching server attempts the open at the file server. If that open fails, the caching server first returns the result to the application, then reads and caches the directory. For each entry in the directory, it adds a node to the naming tree, if not already present, and caches the descriptor in it; when it finishes reading the directory it sets the childListValid flag to indicate that the directory is cached. Replying before reading the directory avoids increasing the latency of the failed operation by the time required to read the directory. Caching only the directories where lookups fail is quite effective: the majority of failed lookups result from path searches, in which an application attempts to open a file in each directory in a list (the search path) until it is found. Because paths are used repeatedly, the names that are not found are clustered in a small number of directories.

The caching server could instead add a *negative* entry to its tree for each name that it looks up but does not find, but caching entire directories is much more attractive than keeping negative entries. First, caching the directories provides descriptor information that can be used to satisfy other requests. Second, it yields a lower miss rate, because once a directory is cached any name not found there can be handled locally, whereas a negative entry covers only the same name; caching the directory amounts to prefetching data for the search path. Third, the number of names not found and the number of entries in their directories are comparable, so that the two alternatives require roughly the same amount of storage. Finally, it is simpler for the cache and file server to maintain coherence if they need deal only with names that do exist.

Partitioning descriptor information. Access to the attributes within a descriptor is not uniform: different operations read and modify different subsets of them. The caching server takes advantage of this fact by partitioning the attributes into a stable and a volatile set that are invalidated independently. The permission information required for most operations is changed only by explicit descriptor writes, which are infrequent, and so is included in the stable set. The volatile set includes attributes, such as the size and last-modified time, that are changed as a side-effect of more frequent operations, including commits (for files) and file creation, deletion, and rename (for directories). While the stable attributes alone are adequate for the caching server to perform many operations, both sets must be valid for the cache to be able to satisfy a read of the descriptor.

When the stable attributes are invalidated for a node, the valid-permissions flags on all file versions on that node are cleared, and the table of users is cleared on all descendants of the node. The table of users must also be cleared for a node and all of its descendants when the node is renamed. Clearing the user tables requires touching all descendants of a node, which might appear to be expensive. However, both changes to stable attributes and the renaming of directories are much less frequent than the name lookups supported by the table of users. Also, the cost of traversing the in-memory tree is low, and it contains only those descendants for which some data is presently cached.

Comparison with interpreting directories. The caching server could cache directories in the same manner as files, and then perform name lookups by searching within the cached directories. While this approach seems to require a minimum of mechanism, several problems undermine its apparent simplicity.

The first problem is one of security: resolving names by searching directories requires trusting the cache to enforce some restriction on access. In particular, when a user has permission to look up names in a directory but not to read it, the server cannot enforce the restriction, since the cache must be allowed to read the entire directory.[8] The problem of security is even greater when importing a foreign file system that imposes its own access controls, using a different model from the native system. In contrast, the approach to caching names used by the caching server does not require that file servers trust caches to protect data.

A second problem with resolving names in this manner is that it presupposes that

[8] In principle, a user who does not have permission to read a directory but is allowed to look up names could determine the directory's contents by enumerating the possible entries and attempting to look up each; in practice, though, the difference in authorization is significant.

	fsbuild	afsbench	latex
read	3284	1495	872
write	9641	2035	124
commit	251	260	8
naming read	663	348	94
misc.	512	271	16
coherence	10638	2302	158
total	24989	6711	1272
traffic ratio	22.2%	19.9%	56.5%
relative to tmp support	101%	90.5%	106%

Table 3.11: Traffic with descriptor caching.

a directory can always be read. But servers in V are allowed to implement directories for which the result of an open (or other operation) is computed using the name as an argument; it might not be possible to enumerate such a directory. The approach used in the prototype supports such servers within the same framework as it does those with more traditional directories.

The final problem with resolving names on clients is that it makes name lookup a compound operation: the client separately fetches or checks the coherence of the directory for each component of the name. As a consequence, operations by name are not necessarily atomic. Imposing concurrency control to ensure that operations are atomic adds complexity and overhead.

Performance. Simulation of caching descriptors and directories yields the traffic levels in Table 3.11 and Figure 3.11. Read-only access to descriptors is therefore reduced by 48–94% compared to that for the basic cache plus temporary support.

Total traffic, however, increases for two of the traces, and decreases only slightly for the third. The reduction in naming reads is offset by the fact the most of the operations saved still require a coherence check. The coherence checks in this case are less expensive than those in earlier traffic measurements: in the earlier measurements, each coherence operation includes a check of permission to name and open the file, while here it does not. Caching descriptor information pays off significantly, then, only if the need for a coherence check can be eliminated for a significant number of these operations.

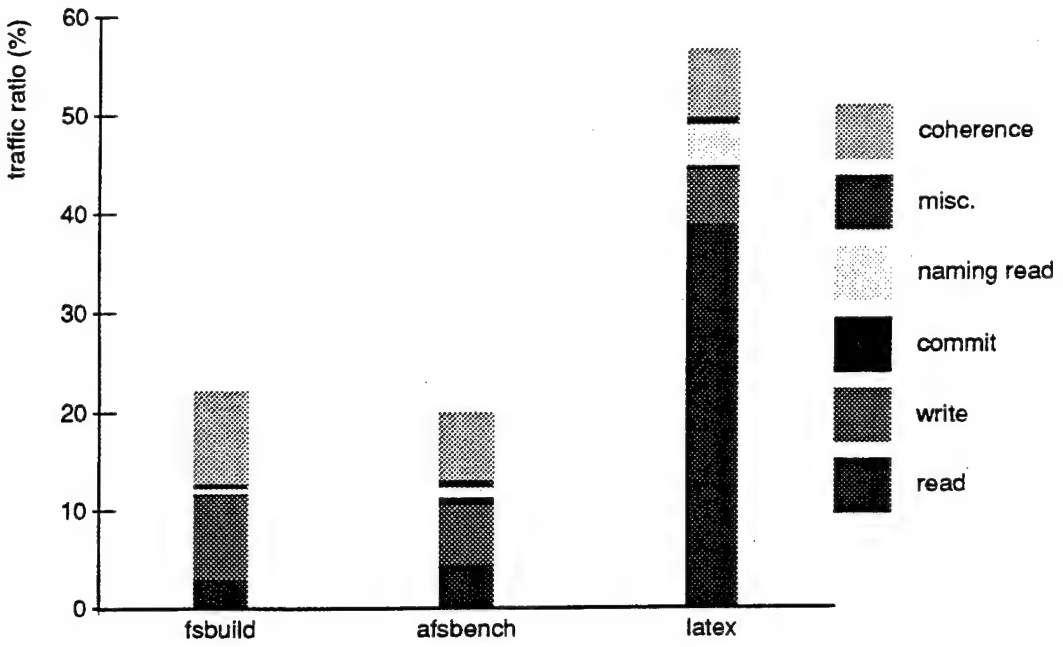


Figure 3.11: Traffic with descriptor caching.

3.5 Using leasing for coherence

3.5.1 Performance

Leasing provides an answer to the need to reduce the number of coherence checks. The analysis in Chapter 2 shows that a term on the order of ten seconds greatly reduces traffic, while also bounding the added delay when a failure occurs. Because the traces contain no writes to shared data, the only coherence traffic is lease extensions. A simple simulation with all files covered by a single lease with a term of ten seconds yields the traffic shown in Table 3.12 and Figure 3.12; the number of coherence checks is reduced by 89–95%.

Just as caching descriptors does not by itself reduce traffic, so also leasing cannot reduce traffic without some caching of descriptor information. If only the contents of files are cached, then each open request requires a request to the server in order to look up the name and check for permission. The net effect of leasing without caching naming and descriptor information is an increase in total traffic: leasing alone does not reduce the number of requests for opening files, and it requires additional traffic to approve updates. In combination, though, leasing and caching descriptors yield a reduction in

	fsbuild	afsbench	latex
read	3284	1495	872
write	9641	2035	124
commit	251	260	8
naming read	663	348	94
misc.	512	271	16
coherence	508	123	17
total	14859	4532	1131
traffic ratio	13.2%	13.5%	50.3%
relative to basic cache	39.5%	47.8%	94.1%
relative to tmp support	60.2%	61.1%	94.1%

Table 3.12: Traffic with descriptor caching and 10-second leases.

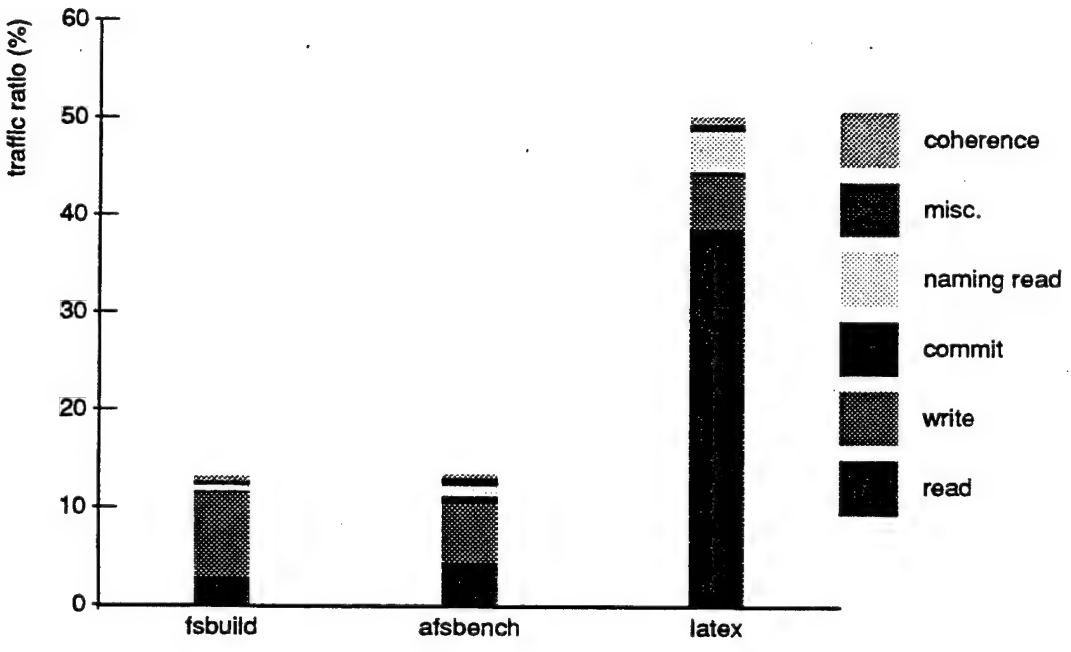


Figure 3.12: Traffic with descriptor caching and 10-second leases.

traffic of up to 40% relative to the basic cache with temporary support.

3.5.2 Implementation details

The general description of leasing in Chapter 2 omits many details that are important in implementation. This section describes how leasing is implemented in the prototype caching server, beginning with a description of *file groups*. It then describes the data structures and processing by which clients keep track of update operations within each file group. The section concludes with improvements suggested by experience in implementing the prototype.

File groups

The prototype maintains coherence in terms of *file groups*, similar to the notion of a *volume* in the Andrew file system [35]. A file group is simply a subtree of the global name space, and a group is identified by the name of its root node. Groups are defined by the administrator of the file server as part of the server's configuration. For example, a public subtree of installed files might form one group, and the subtree below a user's home directory another. Clients learn about groups as a side-effect of coherence requests: a client adds a request for a lease to an operation, and the server's reply includes the prefix of the relevant group as part of the reply.[9]

In the prototype leases are managed at the granularity of an entire file group. Coherence for each file group is maintained separately, but lease requests or replies for multiple groups can be batched into a single message. Because the processing for each group is independent, though, the description that follows is in terms of a single group.

Keeping track of updates

The prototype's implementation of coherence employs two data structures in addition to the records of leases at both the server and clients: a log of updates to each file group, maintained by the server, and a list of pending update operations, maintained by each client. The log provides the support needed to be able to extend a lease after it has expired and to enable clients to disambiguate the order in which operations are performed; the list of pending updates simplifies the client's processing of the log.

[9]The V file system is restricted to a tree, on top of which symbolic links are also supported. In a file system that allows one link to a file, it would be necessary to restrict links to being within a single volume, as is done in AFS.

Some lease-related information is added to each message exchanged by the caching server and file server. The client piggybacks a request to extend its leases on each message it sends, and the file server piggybacks a lease extension on each of its messages. In addition, to requests for update operations the client adds a tag value by which the operation is identified while obtaining approvals, and the server adds a sequence number by which it can be correlated with the log of updates.

When requesting approval of an update, the server sends to leaseholders the operation's request code, the names of its operands, and the tag value from the operation's original request. A client uses the tag to identify the operation in its reply.

A client can request extension of a lease that has expired, in which case updates might have been performed in the interval between the lease expiring and the extension being granted. In order to extend the lease, the server must be able to tell the client which data items have been modified. The prototype handles this information in the form of a log of the update operations that have been performed on each file group. Each record in the log denotes a single update operation and consists of a sequence number, the operation's request code, the names of its operands, and its tag.[10] The client combines reading the log with each request to extend a lease: the client includes in the request the highest sequence number that it has read from the log, and the server includes in the reply any log records with a higher sequence number. To process the reply, the client

1. skips, based on sequence number, any log records that it has already processed,

2. processes each of the remaining records in order, invalidating cached data as required, and then

3. updates is record of the lease with the new expiration

In some cases, the client can use the information in the log to modify, rather than invalidate, cached data. For example, upon reading a record for a rename operation, the client can update its local naming tree instead of invalidating all data associated with both the source and destination names. Updating the cache using the log avoids unnecessarily discarding data from the cache, and it allows the client to keep current the naming information for any files that are cached and the list of children for any directory cached in its entirety.

[10]Recall that, in V, block-level read and write are not relevant for coherence, because read occurs logically when the file is opened and write occurs when the file is closed.

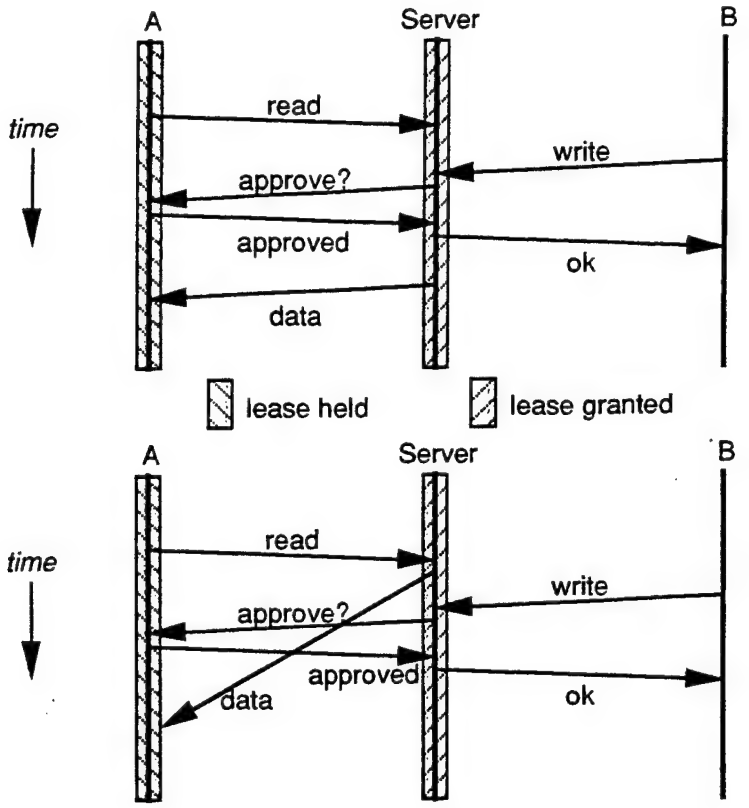


Figure 3.13: An uncertain ordering of operations.

A client also relies on the log in order to ensure that it processes conflicting operations in the same order as the server. The log is needed for this purpose because the order in which messages are received by a client can be insufficient to determine the order of the corresponding operations. For example, consider the sequence of events for a client that reads a file at the approximately the same time that another client is writing it:

1. Client A sends a request to open file F for reading.

2. A receives a request to approve a commit to F by client B.

3. A sends approval of B's commit.

4. A receives the reply for its open request.

As Figure 3.13 shows, this sequence of events is consistent with either ordering for the two operations. But A's next action depends on which order is correct: if the write was performed first, as in the upper diagram, A should cache the newly read data, but if the read was first, as in the lower diagram, the data should not be cached.

This second case in Figure 3.13 also points out that when a client approves an update, it cannot simply invalidate the affected data and forget about the operation; if it does, it might apply operations to its cache in the wrong order. In order to ensure that each operation is applied to the cache exactly once and in the correct order, the client does not process any operation except when it reads the corresponding record from the log. The client correlates the results of operations with the log entries using the sequence number that the server includes with the result. For a write, the sequence number matches the log entry for the update; for a read it matches the immediately preceding update.

There can be a significant interval between a client approving an update and the same client reading the corresponding record in the log. During that interval, the client does not know whether the operation has been performed, but the affected data is still marked as valid in its cache. To prevent it from using the data in this period of uncertainty, the client adds each a record for each update that it approves to its own list of pending updates; the record includes the operation's request code and its operand, along with the operation's tag. Before using cached data to satisfy any read request, the client checks for a conflict between the requested read and each operation on the list of pending updates. If a conflict is found, the client must query the server to determine whether the operation has been performed; it can do so by requesting an extension to its lease.

The tag for each update operation provides the basis for keeping track of approved updates. When a client requests an operation that might produce an update, the client adds a unique tag value to the request, and it adds the operation to its list of approved updates.[11] The server includes the tag when it requests approval of the update from another client; that client adds the operation to it list, then sends its approval, identifying the approved update by its tag. A client removes an operation from its list when the corresponding tag appears in a log entry.

Not every update that is approved actually results in a change. An update request can be rejected by the server if, for example, the requesting user is not authorized to perform it. Also, the requester sometimes does not know whether an operation will produce an update: when opening a file for writing, the file is created only if it does not exist. The file server inserts such an operation in its log of updates, but with a request code of `NO_OPERATION`. The only action taken by the client upon reading a `NO_OPERATION` log record is to remove the operation with the matching tag from its list of pending operations.

[11] The tag is included for operations requested by caches. The file server supplies a tag if the operation is requested via the normal application interface.

The cache has to be prepared to deal with failures when handling coherence messages. A cache might encounter a discontinuity in reading the log, either because the server crashed, and it stored the log only in volatile memory, or because the server discarded old log entries before the cache had opportunity to read them. When the cache detects that it may have missed some log entries, it must invalidate all cached data for the file group if the lease has expired; if the leases has not expired, the cache discards any data affected by the operations in its list of approved updates. To avoid missing log records with limited memory devoted to the server log, the cache periodically queries the file server, extending all of its leases to the present time, in order to read the log; this period is presently set at five minutes.

The cache also has to be able to clear its list of pending updates. To allow it to do so, each lease extension message from the server includes a list of the pending update operations that the client has approved. After performing invalidations required by a discontinuity in the log, the cache rebuilds its list of approved updates from the list in the extension message. An operation for which the client receives no reply must be left on the list of pending updates until either its outcome is learned from communication with the server or all data it might affect has been invalidated in the cache.

Possible improvements

The prototype caching server includes the lease support described here, and the bulk of it has been exercised with test scaffolding. At this writing, a file server that supports leasing has been only partially implemented. Experience with the prototype suggests several improvements that could simplify the implementation of leasing for both the cache and file server and also improve performance when there is sharing.

Shorter identifiers. The use of character-string names throughout makes the prototype more complicated and less efficient than it might otherwise be. Elsewhere in V, character-string names are the only persistent identifiers for named objects; the design attempts to apply the same approach to caching, but it proves awkward. Managing storage for variable-length strings is clumsy in C and C++, and frequent interpretation and comparison of names is inefficient. The character-string names also inflate the size of log entries. In an improved implementation, the server would assign a fixed-length identifier to each node within a file group, and that identifier would replace the name in most coherence messages. In a reply, the server would send along with character-string

name for the path an array of the identifiers corresponding to each of the components; the cache thus learns these identifiers as a side-effect of its operations. The identifiers replace names in most log entries; a file commit, for example, would need just the identifier. For file deletion, the identifier of the parent is included along with that of the deleted node. The creation of a node would specify the parent's identifier, that of the new child, and the name component of the new child. A rename requires four identifiers and one name component: the node moved, its old parent, its new parent, and the node it replaces, if any, plus the component of its new name. Because they are local to a file group, identifiers can be short, reducing the size of log entries in addition to simplifying handling.

Finer grain for leasing. The prototype manages leases at the granularity of entire file groups, which is adequate if write-sharing of file groups is not common, as in the traces. We anticipate more sharing in the future, however, if a user is able to employ a set of hosts over an extended period. Under this pattern of sharing the directories in the user's working set are write-shared among those hosts, and that files within those directories are also sequentially write-shared. To support such sharing well, the prototype needs to be extended to allow the coverage of a lease to be specified at a finer grain than the entire file group.

With short identifiers, it becomes practical to list the nodes in a file group that are covered by a lease; the attributes for each node could be specified by an additional four bits, one each for stable attributes, volatile attributes, file contents, or list of children.

Relinquishing leases. In addition to adding small units to a lease, the cache needs to be able to indicate that it no longer holds a copy of (or an interest in) a data item. For example, in response to a request for approval of an update, the cache might invalidate a directory or descriptor that has not been recently accessed, and relinquish its lease so that it is not required to approve future updates. Each leased operation, extension request, or approval of an update includes a list of identifiers over which the sender relinquishes its lease.

Relinquishing a lease produces the same sort of ordering problem as approving an update. When a cache sends a message relinquishing its lease over some item, that message might be received by the file server shortly after it has sent a reply (or other message) that extends a lease over that same item. The correct order can be established by treating the relinquish as an update: the client tags the request and adds it to its list of

pending updates, and the server inserts the relinquish into the log with the request code NO_OPERATION. Unlike other NO_OPERATION records in the log, the cache must recognize when it removes a relinquish operation from its list of approved updates and modify its record of lease coverage.

Partial invalidation for file contents. When a file is updated, the prototype discards all blocks cached from an old version of it. Data is needlessly invalidated if only a few blocks of the file change, or if data is added only to its end. For the applications run in V, this waste is not a problem—programs almost always write a file in its entirety. The same is true for most Unix software, too. Other applications (in the future) might not have this property; database updates commonly change only a few blocks in a large file.

Extending the coherence support to handle invalidating only part of a file is straightforward: all that is needed is to add to the log entry for a commit of a file version a list of block ranges that are modified. The memory server interface, however, would need extensions to allow partial invalidation from within an open file.

Write-broadcast for descriptor data. The most common operations change just a couple of fields in a descriptor, such as the modified time and length. The new values for these fields could be added to the log entry, effectively changing the caching of descriptor information from write-invalidate to write-broadcast. Further study would be needed to determine whether the reduction in cache misses on descriptor data is worth the increase in the size of log records.

3.6 Additional issues for caching in V

3.6.1 Limitations

We have not considered here any details of block-level caching; the only assumption we have made is that the cache is large enough to avoid any replacement. Effective caching does depend on a number of matters at the block level, including cache size, the size of transfers, replacement policy, and prefetch policy. For caching in RAM, the interactions between file caching and virtual memory management must all be considered. These concerns have been explored by others; Chapter 5 surveys this work.

The prototype does not address some issues of integrating file caching into V. In practice, the most apparent weakness is that caching is not transparent with respect to

naming, and so caching does not work well with remote execution and program migration. In addition, the prototype performs poorly when a files is opened at one host and then read or written at another, a situation that arises primarily from remote execution. Under these circumstances, access should be via the cache co-located with the reader or writer instead of the cache where the file was opened, but V presently has no mechanism to support such a rebinding. Both of these problems—naming and rebinding of open files—are fundamental to replication of data, not just caching, and solutions found in the more general context should be applied to caching as well.

A full consideration of the implications of caching for file-server design is also beyond the scope of this dissertation. With effective caching, writes outnumber reads; so writes must be handled efficiently. Good response time depends especially on having low latency for operations that commit data to persistent storage, which include changes to directories and descriptors as well as commits of file versions. Logging updates to metadata, as in Hagmann's reimplementation of the Cedar file system [33], reduces latency by reducing the amount of seeking on disk. Latency can be further reduced by using nonvolatile RAM for the tail of the commit log, or by placing the log on either a separate disk or, in a disk array [54], on a separate set of heads.

3.6.2 Performance

Sharing. Because the traces include no write sharing, they generate no coherence traffic other than lease extensions. Under the style of sharing anticipated in Section 3.2.2, lease conflicts, and the attendant update-approval traffic, should be common, but only for small numbers of leaseholders. Most of the updates affect directories, and the sharing could lead to thrashing if the entire directory were invalidated on each update. The partitioning of descriptor contents, in Section 3.4, and the use of the log to update cached directory information, in Section 3.5, both serve to minimize invalidation of descriptor and directory information.

Response time. The performance evaluation presented in this chapter has been almost exclusively in terms of traffic; response time has been largely ignored. The primary contribution of the file service to response is the delay when an application must wait for an operation to be performed at the file server. Much of that delay is from congestion at the file server; it is minimized by reducing total traffic, which the design here does, and by handling operations efficiently at the file server, which is outside our concern here.

There are three kinds of operations on which an application must wait: cache misses of reads, commits, and coherence checks.

Cache misses are generally minimized by reductions in total read traffic, but they are further lowered by prefetching data. Block-level reads dominate the read-only traffic; prefetching for these is the responsibility of the memory server. The caching server handles naming reads, and caching the entire directory after a name lookup fails is an effective way to prefetch naming information for path searches. Only 23–46% of the naming reads performed are cache misses, and majority of those are to open files not yet cached.

The number of commits required is significantly reduced by the support for temporary data, and those that remain are unavoidable if file storage is to be reliable. Fortunately, commits are only a tiny fraction of the total traffic in our measurements. The delay for commits depends mainly on the design of the file server, which needs to handle both writes and commits efficiently. Commits of shared data also require approval of the update, but the time to do so is in most cases dominated by the time for writing data to disk.[12] The delay for a commit is therefore not more than a few tens of milliseconds, while the application engages in an average of 6–30 seconds of processing or other activity per commit; the delay added in order to reliably commit data is on the order of only one percent of the response time.

With leasing, the rate at which coherence checks are required is extremely small: at most one check is required per term. These checks therefore account for much less than one percent of the response time.

The caching server does several things in order to minimize response time: it reduces total traffic, and therefore congestion; it avoids many cache misses on naming reads by prefetching entire directories; it reduces the number of commits by its support for temporary files; and it limits the delay to provide coherence to a very small fraction of total response time. There is little more that the caching server can do; response time depends to a large degree on the memory server and file server.

3.6.3 Security

In the prototype, the caching server does not require any special privilege from the file servers; in particular, the file servers do not have to trust a cache to enforce access

[12] In the absence of failures; when a client crashes, the delay is bounded by the lease term. But such failures are infrequent enough that they do not affect average performance.

controls. The only special privilege that the cache requires is the ability to act on behalf of applications that make requests of it; *i.e.*, that it be able to present their credentials to the file server. A user must, of course, trust a cache he is using to not abuse this privilege, in the same way that he must trust any software he uses.

One aspect of maintaining security that we have not dealt with in full detail is that of avoiding inadvertent disclosure through the coherence messages. A cache should see only the union of what its users are authorized to see. The coherence log and pending operations therefore must be filtered on a per-cache basis to avoid disclosure. The cache must therefore register with the file server the users on whose behalf it is making requests. The file server can collect these implicitly from the `LEASED_OPERATION` requests it receives, but there also need to be explicit requests available to the cache to add or remove a user from the set, plus the means for the file server to challenge the cache to again present the credentials for the users on whose behalf it is working.

3.7 Applying the results to other systems

This chapter has evaluated for performance the file service in V; there are two kinds of differences that could limit the applicability of the results here to other systems: differences in the semantics of the operations and differences in the pattern of access.

Semantics. The most significant difference between V and many other file systems is that it provides atomic open files. This affects performance in two ways: it reduces the number of operations that require synchronous writes in order to guarantee recoverability, and it lowers the number of operations that are significant from the standpoint of coherence. With some adaptation, however, the techniques described in this chapter work just as well in systems that make data visible at the granularity of individual read and write operations.

A guarantee of recoverability can be defined that requires very nearly the same number of synchronous writes as for V's atomic open files. Specifically, the file service can guarantee that data has been written to nonvolatile storage whenever:

1. The writer closes the file,

2. The writing program otherwise commits the data,[13] or

[13]Some versions of Unix provide the `fsync` call for this purpose.

3. The data is read by a program other than the writer.

The first two cases correspond to the guarantees in V, and they provide the writer assurance that the writes will persist. The third case is necessary to guarantee to readers that the data they read will also survive; it causes a read operation to incur added delay only when it reads as-yet uncommitted data. This case occurs only when a file is concurrently write-shared, which is very infrequent. The total number of synchronous writes is thus very close to that in V.

Because the number of operations actually requiring commits is comparable, the support for temporary files should be just as effective in reducing both traffic and delay. Because writes to other files are not synchronous, however, the cache must be write-back rather than write-through. In this case the data in the cache is not yet committed; so it does not have the reliability problems noted in Section 1.2. Readers see only committed data, and writers learn of failures when they close or otherwise commit the file. Leasing for write-back caches is described in Section 2.4.3.

The semantics for files in V produce low rates of reads and writes from the standpoint of coherence, because only opens and closes are counted. Most files are open, though, for much less than ten seconds, so that a single lease term of that length will usually cover all of the access while a file is open. The higher rate of writes can be handled by making the cache write-back, as described above, and having all caches but the writer relinquish their leases. The only case in which the higher rate of writes is still significant is when a file is concurrently write-shared. Turning off caching during concurrent write-sharing, as is done in Sprite [50], avoids extra traffic to approve writes; with leasing, caching is easily turned off by using a term of zero length. As long as concurrent write-sharing is rare, which measurements show it to be in Unix systems [65], the resulting traffic differs very little from that for V.

Under these conditions, separate support for temporaries still yields a reduction in traffic comparable to that achieved by delayed write. With recoverability guaranteed as above, the number of commits—and the delay they incur—is small. It is possible to have good performance without sacrificing robustness.

Access patterns. There are two reasons that access patterns in other systems might differ significantly from those in the traces from V: the way that system services are implemented and the applications running on the system.

None of the file access captured in the traces is for access to system services, since

services in V are accessed by communicating directly with a server program, not by reading or writing shared files. Most of the write-sharing observed in Unix systems, though, is of files that serve as the interface to services such as printer spooling, user information, system status, mail or news [11, 26, 65]. In such a system, there is more write-sharing of files than is observed in V, but that sharing still represents a tiny fraction of all file access. The result is only a small degradation in overall performance.

One activity missing from the traces is access to large databases, which would produce different patterns of both operations and sharing. The details of the access pattern depend on the structure of the interface to the database, including whether data is accessed via shared files or through a database server. Generally, though, database access would probably increase the amount of sharing, and it would make partial invalidation of file contents more important than it is in the traces. Two of the issues arising in supporting database access, caching structured data and supporting transactions, are taken up in the next chapter.

3.8 Summary

This chapter has described a prototype cache for file service in the V distributed operating system and evaluated its performance in terms of traffic based on traces of file access. Three enhancements to the basic cache design boost its performance without compromising the reliability, availability, or coherence of the file service.

First, special support for temporary data reduces write traffic by 46-54% in the traces that use temporary files, and it also eliminates coherence and committing operations for these files, for an overall reduction in traffic of 20–35%. Special handling for temporary files yields a gain in performance comparable to that obtained by delaying for thirty seconds writes to all files. Unlike delayed write, though, support for temporaries obtains improved performance without sacrificing the reliability of permanent files.

The second and third improvements, caching of descriptor information and using leasing to maintain coherence, are effective only in combination; together they reduce traffic by as much as 40%. Caching information from descriptors allows the caching server to handle name lookups and permission checks for cached data, and caching entire directories enables it to also handle lookups of names that do not exist, which are quite common. The caching server reduces the frequency with which it needs to read descriptors and directories by caching the results of name lookups; it reads an entire directory only

when an application requests it or when a name lookup fails. Also, the caching server avoids having to refetch data by invalidating only part of the information in a descriptor in response to most updates, and by using the log of coherence information to update, rather than invalidate, cached directories whenever possible. The use of leases with a ten-second term eliminates the need for a coherence check on most reads of cached data.

In combination, these three improvements to the basic cache reduce traffic by 60% and 52% for the `fsbuild` and `afsbench` traces. The remaining trace, `latex`, improves only 6% overall, but this is because over 75% of its traffic is reads that no cache can avoid.

While the evaluation here has focused on traffic, the reductions in traffic should produce corresponding reductions in queueing delays at server CPU and disk, yielding good response time. Given the traffic remaining from the cache, low delay depends on the file server handling writes and committing operations with low latency, and on effective prefetching by the block-level cache.

Chapter 4

Additional uses for leasing

The preceding chapters have focused on using caching to improve the performance of a traditional file service. This chapter expands the scope of leasing beyond that of the V-system file cache in the preceding chapter. It develops leasing in three directions:

Scaling up. The prototype increases the number of clients that a server or network can support by reducing the traffic per client by a factor of up to seven. Section 4.1 looks at how secondary caches can be used when a system grows very large or spreads over a wide-area network.

Other storage services. The file service that the prototype supports is only one possible storage service, and a fairly primitive one at that. One might ask whether the techniques for caching described here could be applied in other storage services. An important feature of many more sophisticated storage services, such as databases, is support for atomic transactions. Transactions are a useful extension to even basic file service, particularly in a distributed system, since they simplify the development of robust applications. Section 4.2 considers problems in making caching efficiently support transactions.

Availability. In the description of leasing, availability was considered as a constraint: the availability of the file service had to be preserved at the level provided in the absence of caching. But for some cases the availability provided by a single file serve is not high enough. Section 4.3 considers two routes to increased availability: replication of servers, which was ignored in Chapter 2, and taking advantage of cached data by trading coherence to gain availability.

4.1 Caching in very large systems

Chapter 1 identified a trend toward large systems—either in number of client hosts or in geographical scope—as a motivation for caching. A large system typically consists of clusters of hosts on local area networks, with the clusters connected by a wider-area network.

The prototype cache improves scalability in both senses. A larger number of clients can be supported by a server or network, since the traffic per client is reduced to as little as 13% of that without caching, or 40% of that with the basic cache. Synchronous requests—read misses, coherence checks, and commits—incur delay for round-trip communication; the prototype especially reduces the frequency of these requests.

Scalability can be further improved by using a secondary cache to mediate a client's access to file servers outside the local cluster. The secondary cache is very similar to the prototype caching server in the preceding chapter, except that it is responsible for the portion of the naming tree corresponding to file servers outside the cluster. Secondary caches allow inter-cluster messages for reads and coherence to be amortized over multiple clients within a cluster, reducing both server load per client and communication delay per operation. Secondary caches also reduce the number of clients for which a file server must keep state, because the server is accessed directly by (and grants leases to) only clients within its cluster and secondary caches in other clusters.

Coherence within a cluster is handled by *subleasing*. Once it holds a lease from the remote file server, the secondary cache can grant subleases to its clients. The term of a sublease must be contained within that of the lease under which it is granted. Also, the secondary cache cannot approve an update or relinquish its lease until it has obtained the same action from the holders of its subleases.

While the prototype caches only in RAM, a secondary cache could benefit more from also caching on disk. The secondary cache can then be larger, to yield a higher hit rate. Also, the latency to access a local disk compares much more favorably to the latency of a wide-area network than to that of a local-area network. Also, the speed of writes to the cache's disk is not critical to performance, since data needs to be written to disk only when it is not invalid but would otherwise be discarded; most access would still be from RAM.

When the clusters are not tightly coupled, but instead form a federation of autonomous systems, the secondary cache serves as a file-service gateway. One function of such a gateway is the translation of identifiers, such as those for users, between the

systems. In the naming design for extending V to wide-area use [16], the gateway's function is combined with that of a liaison server that resolves names outside the local system. From the standpoint of access control, the prototype's strategy of caching results works well for the gateway, too, since the cache does not have to understand the policies of the remote system, nor must the remote file servers trust the cache to enforce protections.

In summary, secondary caches further increase the scalability of a caching file service, and the approaches used in the prototype, including leasing, work well for secondary caches.

4.2 Caching and atomic transactions

Many applications of storage services that benefit from atomic transactions would also benefit from the improved performance offered by caching. Two parts of the support for transactions require adaptation to work efficiently with caching: concurrency control and atomic commit processing.

4.2.1 Concurrency control

For a system with caches, coherence defines what is a correct logical order for read and write operations in order to ensure that caching does not affect the results of operations. Transactions typically impose on the order of operations the constraint of *serializability*: the logical order must be equivalent to one in which the operations in separate transactions are not interleaved. Cache coherence and transactional concurrency control could be implemented independently, but then there would be two mechanisms attempting to impose an order on operations. At best the result is duplicated effort; at worst the attempts conflict, since the two orders are not necessarily the same. For transactions, then, concurrency control takes the place of cache coherence.[1]

Leasing can be applied to concurrency control in much the same way as to coherence.

[1] Actually, the notion of coherence is not completely displaced. Non-transactional access can be viewed as a series of degenerate transactions, where each consists of a single operation and commits when the operation's result is returned. Such a system is trivially serializable: operations can be executed in any order at all, as long as each is atomic. Coherence constrains the order of these transactions: if one transaction is observed to commit before a second is initiated, then the first transaction must also precede the second in the serialization order. The notion of coherence as an additional constraint on serialization order generalizes to compound transactions.

That serializability allows transactions to execute "out of order" is commonly overlooked because two-phase locking, the most commonly used technique for concurrency control, does not allow such anomalous behavior.

In particular, a lease can allow a cache to grant locks locally, so that a lock can be claimed and released repeatedly without communication with the server. A cache can grant read or write locks when it holds a lease of the corresponding mode. The cache cannot relinquish the lease, allow it to expire, or approve a conflicting update while the lock is held, however, or the lock must be broken and the transaction aborted.

The fact that transactions can be aborted, though, allows a cache to grant a lock without holding an unexpired lease, at the risk of having to later abort the transaction if an update occurs. In particular, if a lease has expired, the cache could optimistically grant the lock before seeking an extension, so that execution of the transaction does not have to wait for the server to grant the extension. The cache can also allow a lease to expire while a lock is held. Before the transaction commits, though, the cache must extend the lease, aborting the transaction if any conflicting update has occurred. With a limit on the period for which the lease is allowed to have lapsed, this approach could be described as boundedly optimistic concurrency control.

Additional support is required for write locks when a transaction might access an item through more than one cache. When the item is accessed through the first cache, that cache obtains a lease and grants a lock; later, the same transaction, executing on a different host, tries to access the same item through a second cache, and the lease requested by the second cache conflicts with that held by the first. To handle this case, each request for a lease needs to include an identifier for the transaction, if any, that is requesting the lock, and the file server includes the transaction's identifier in its request that the first leaseholder relinquish its lease. If the requesting transaction is the same one that holds the lock,[2] then the cache relinquishes its lease, but indicates that the item is locked by the requesting transaction. The file server records the write lock[3] as held by the transaction, then grants the requested lease. Any item that is write-locked and shared has the lock recorded at the file server; there can be one write lease or many read leases over the item, but all leaseholders know the identity of the transaction holding the lock.

Multiversion concurrency control methods, surveyed in [10], are a natural fit with caching, since the caches do cause additional version to exist, whether or not the server maintains multiple versions. Read leases, in fact, correspond quite closely to multiversion read locks, and the collection of approvals for an update to upgrading write locks to certification locks. Granting either read or write locks for an item should be avoided

[2]Or, for nested transactions, the requester is a subtransaction of the one holding the lock.

[3]For there to be a conflict, one of the requests had to be for a write lock.

while an update is pending against it, since the transaction receiving such a lock is likely to conflict with the one waiting to commit, and one of them will have to be aborted. This contrasts with the non-transactional model in Chapter 2, where reads are allowed while an update is pending; the difference is that there each operation is a separate transaction, and the read "commits" immediately with no possibility of conflict with the pending write. Under multiversion locking, an updating transaction must retain all of its locks—and hence the cache its leases—until commit, while a read-only transaction can release them once the last read is performed:

Under multiversion locking, read and write locks do not conflict; so a cache can still hold a read lease for an item that is write locked. The cache must know the identity of the transaction holding the write lock, however, because reads by that transaction return a different version. If an item is not shared, read and write leases can be used as before, giving the right to grant both read and write locks. When an item is shared, however, only read leases are used for the committed version, with the leasing over any uncommitted version managed separately. All holders of read leases on the committed version must approve the granting of a new write lock, at which time they and the file server record the identity of the transaction holding the lock. When the new version is committed, approval is again required from the caches holding leases over the committed version, which invalidate their old copy of the committed version and discard their record of the lock.

Other concurrency control methods can also be adapted for caching and leasing, including timestamp-based and hybrid methods. Some special care is required in selecting timestamps, especially for multiversion methods, in order to preserve coherence between transactions; the details of adapting these methods are not included here.

4.2.2 Atomic commit processing

A transaction that involves more than one server must employ an atomic commit protocol to ensure that all of the participating servers come to the same decision about whether the transaction commits or aborts. In order to accommodate caching, existing protocols for atomic commit can be extended to cope with leasing and to allow caches as participants.

As an example, consider two-phase commit [31]. One of the sites involved in a transaction is designated as the *coordinator* for the transaction. When the transaction attempts to commit, the coordinator sends a *prepare-to-commit* message to the other participants. Each of them writes any modified data to stable storage and logs the prepare-to-commit

before replying *prepared* to the coordinator. After it has received a reply from all of the participants, the coordinator sends them a *commit* message. Each participant then logs the transaction as committed and releases any locks it holds.[4]

If for some reason a participant is unable to commit the transaction, it replies to the prepare-to-commit with an indication that the transaction must abort. Upon receiving such a reply the coordinator sends an *abort* message to all participants, which log the transaction as aborted and abort its local effects. The coordinator can send an abort message at any time before it sends a commit; it does so, for example, if a participant fails to respond in time to a prepare-to-commit.

From the time it sends a *prepared* message until it has logged either a commit or an abort, a participant is uncertain as to the transaction's outcome. Locks must be held through this in-doubt period. Also, a failure during this period requires additional effort during recovery to learn of the outcome, and leaves that transaction blocked, with its locks held, until the participant is able to learn of the outcome. Because a transaction is blocked while in doubt, and because of the additional costs for recovery from failures, it is desirable to keep the period of uncertainty short.

Leasing and atomic commit

Leases can lengthen the in-doubt period, because a transaction cannot commit while there is still a lease outstanding against data it writes. A server participating in a transaction could ensure this by delaying its reply to the prepare-to-commit until it has obtained approval for each lease that has not expired. If this delay is significantly longer than the time to perform the required writes to the log, the period during which other participants are uncertain of the transaction's outcome is lengthened. Also, if the delay is too long, the coordinator will time out and abort the transaction. When a leaseholder has crashed or is unreachable, the delay can be up to the term of the lease, which may be a long time.

The period for which locks are held cannot be reduced, but the period of uncertainty can be shortened. Each participant replies promptly to the prepare-to-commit, but includes in its reply the latest expiration time for an outstanding lease. The coordinator includes the latest of these times in its commit message. Participants can then record the transaction to be committed as of the indicated time: its eventual outcome is known, but write locks must still must be held until the commit time.[5] A participant that receives

[4] In single-value locking, read locks can be released when the participant replies to the prepare-to-commit; under multiversion locking they must be held until all of the certification locks are obtained.

[5] For multiversion locking, all locks, including read locks, must be held. The certification locks held

approval for update that lowers its maximum expiration time can inform the coordinator, which can send a new commit message with an earlier as-of time. Participants that receive this message may release their locks at that earlier time.

As in the absence of transactions, the term of leases determines how much a client failure can degrade the performance of other clients. When a leaseholder fails, it can cause a transaction to block for the term of the lease; any transaction conflicting with the blocked one is also blocked.

Caches as participants

An atomic commit protocol requires votes from all of the participants in a transaction, which includes the caches as well as the file servers. The handling of the voting must be modified to allow for caches as participants.

In the first round, the coordinator sends the prepare-to-commit to all participants, including caching servers. File servers vote in the normal manner. When a caching server receives a prepare-to-commit, it writes back any data that is still dirty, with the same transaction identifier, then sends a cache-prepare-to-commit to each file server from which it cached data for the transaction, whether read or written. The file server replies to the cache as it would if the message had come from the coordinator, but on the basis of the additional information about reads and writes that it has received from the cache;[6] a file server that previously replied to the coordinator as prepared can later reply to a cache as aborting. A cache replies to the coordinator as prepared only if all of the writes succeed and each server replies to it as prepared. In order to keep the coordinator from timing out while all this is happening, the cache may need to send it messages indicating that it is not yet prepared to commit.

In summary, caching can also efficiently support atomic transactions on multiple items. Leasing enables the caches to handle many of the locking requests, so that it reduces server traffic for concurrency control in the same way that it reduces coherence traffic in a non-transactional setting. Furthermore, encapsulating a group of operations within a transaction can amplify caching's reduction of response time: each transaction incurs only once the delay for a synchronous write to the server's nonvolatile storage, whereas several operations might require separate commits if they were not part of a single

prohibit both reading and writing by other transactions.

[6]The server can release read locks when it first responds to the prepare-to-commit, but it must maintain information about them until the commit or abort decision is received from the coordinator, so that it can detect conflicts with locks granted by caches.

transaction. The prospects are good for exploiting caching to improve the performance of storage services that provide transaction management.

4.3 Improving availability

Previous chapters have assumed that each file is stored at only one server, and the attention paid to availability has been limited to ensuring that adding client caching does not reduce availability. The level of availability provided by a single server is not always high enough, however, in which case data can be replicated at multiple servers to increase its availability.

This section considers two questions concerning caching and availability. The first is the problem of maintaining cache coherence when data is replicated. The second is how coherence might be explicitly traded off to increase availability.

4.3.1 Caching replicated data

Caching and replicated data

Replicating data offers the possibility of increased availability, reliability, and performance. Caching is a special case of replication with the goal of improving performance, especially for reads; cache coherence corresponds to notion of *mutual consistency* for replicated data. Cache coherence could in principle be handled by simply treating caches in the same way as persistent replicas. There are several reasons, though, that caches need to be treated differently. First, copies in caches are transient, while replication techniques usually assume that the set of copies of an item does not change frequently. The fixed set of replicas is also intended to enhance reliability, which caches do not, since their copies may be discarded at any time. Finally, the number of caches can be very large, while replication is usually targeted at a small number of copies.

A variety of approaches to replication are surveyed in [10] and [19]; leasing can be used for cache coherence in conjunction with several of these methods. How leasing is used depends on the patterns of communication between a client and the servers, and on how the requirements of leasing can be met by the servers. It also depends on the guarantee of coherence made for access to the replicas: some methods, such as *available copies* algorithms, do not tolerate communications failures, and so are not usable in the environments targeted by this research, while other algorithms, such as *virtual partition*, are coherent only with respect to restricted communication, so that a discussion of caching

from them would be dominated by describing the guarantees that caches can make. Two approaches that do fit well with our concerns for coherence and for fault-tolerance are quorum consensus and primary site; these serve as examples of how leasing interacts with replication.

Quorum consensus. Quorum consensus (also known as weighted voting) was introduced by Gifford [28]. Each replica is assigned some number of votes, and votes must be collected for each read or write. A read requires contacting sites for which the total number of votes constitutes a read quorum to determine which sites hold the current version; similarly, a write must be performed at sites with at least a write quorum of the votes.

Leasing is easily combined with replication by quorum consensus: holding a lease in effect caches a server's votes. To know that its copy of an item is current, a cache needs to know that it is the most recent among a read quorum of the replicas. The cache can know this by holding a set of leases covering a read quorum. When it obtains a lease, the cache must also retrieve from each of the replicas it contacts the number of votes, so that it can determine when it has a quorum, and a version number for that replica, so that it can determine which is the most recent version, just as a non-caching client would. Because the term limits the frequency with which votes are required for reading, the high overhead of reads under quorum consensus is greatly reduced. Write-back caching works in a similar manner: in order to hold dirty data, a cache must hold write leases covering a write quorum of the replicas.

The cost of maintaining coherence when caching replicated data depends on the number of servers in the quorums. For example, the load for extending (read) leases is spread among the servers. If there are n_S servers storing an item, and a read quorum contains q_R servers, then a cache must hold q_R leases over an item. The cache requests q_R extensions per term, and the cache also must approve a separate update of each of the q_R replicas when the item is written. The traffic handled by the client and by the network therefore increases with replication. Because each client obtains leases from only a read quorum q_R of the n_S servers, the coherence traffic handled by each server decreases to an average of q_R/n_S extensions per client per term.

In the case of a system consisting of clusters of workstations and servers connected by a wide-area network, subleasing can be used to reduce the non-local traffic, just as for non-replicated data. The gateway cache obtains leases over the remote replicas and participates in local votes on their behalf; in this case, the lease functions as a *proxy*.

If this is done, though, duplicate votes can result if there is more than one gateway in a cluster or if a client communicates with the remote server directly as well as with the local gateway. To prevent this, each server's votes need to be accompanied by an identifier for the server so that the client collecting votes can discard duplicates. The gateway also needs to monitor the status of the local replicas, so that it can obtain leases over additional remote replicas to make up the read quorum when the local replica is unavailable. It can do this either by seeking additional leases in response to a repeated request for votes (leases) from a client, or by maintaining a lease over the local replicas and seeking additional leases when it is unable to extend it.

Primary site

Primary site methods designate a single site to handle all of the concurrency control and version management for a given data item. If the primary site fails, a new primary is elected from among the surviving replicas.[7] Failure of the primary is, of course, detected by timeout: an election must be held each timeout interval, and a site must cease to function as the primary if it not reelected within the interval. The cost of these periodic elections can be reduced by using a simple protocol to reelect the existing primary, resorting to a more general election only when the reelection fails.

Leasing under a primary site scheme is simple: all clients obtain their leases from the primary site, and a client handles coherence exactly as for non-replicated data. The only interaction between leasing and coherence in this case is the restrictions that leasing imposes on the servers: a newly elected primary must honor any leases granted by its predecessor. This can be ensured by having the primary inform one or more secondaries before it grants each lease or extension, just as writes must be made to at least one secondary site in order to prevent them from being lost. The leasing constraint can also be enforced by having the new primary delay writes for the maximum term, just as for a single server recovering after a crash.

Leasing can also be used among the replicas. For example, a secondary site can obtain a lease from the primary and in turn grant subleases to clients. The load for extensions messages can then be shared among the servers instead of being concentrated on the primary; the costs of doing so are additional approvals required for updates and, due to the constraint of subleasing, slightly shorter terms for clients.

[7] Schemes in which a new primary cannot be dynamically selected are degenerate cases of quorum consensus, with all votes held by a single site.

Behavior very similar to the periodic election of a primary site can also be obtained by using leases among replicas in conjunction with quorum consensus. If a server holds leases over a read quorum of the replicas (including its own), then it is assured that its copy is current. On the basis of these leases, the server can grant a sublease to a client; the server can, in fact, grant a single lease over the replicated item rather than a set of leases over the separate replicas, such that the client can ignore the fact of replication in maintaining coherence. A client then needs to communicate with only one of the servers to be able to cache data for reading. Like primary site, this scheme keeps the cost of reading data low, because a client needs to communicate with only one server in order to cache data for reading. It also eliminates the need for a separate election mechanism; lease extensions take the place of voting. The cost for using leases among the replicas is an increased in the degree of sharing, because servers as well as clients hold leases over an item, and therefore increased cost for writes.

Write leases can mimic more closely the centralized behavior of an elected primary site. The election is effected by one site obtaining write leases over a write quorum of the replicas. During the term of those leases, the site holding them controls all access to the covered items, and only that site can grant leases to clients. In comparison with separate election of a primary site, this scheme has the virtue of reusing the same mechanism required to support clients; the management of leases, though, would need to incorporate the techniques from election protocols to ensure that some site does eventually obtain a quorum.

In summary, leasing can guarantee coherence when caching replicated data, and it can even be used among replicas to reduce the cost of read access.

4.3.2 Coherence and availability

With a large enough cache, it is likely that a workstation has cached a copy of the files that its user is working with, such that the cache might be able to provide access to them even when the file server is not available. When the server is inaccessible, though, the cache cannot ensure that access is coherent. Any gain in availability must therefore be purchased by sacrificing the guarantee of coherence.

How the trade-off between coherence and availability is made depends on the circumstances, of which there are two general types. The first case includes applications that function correctly with stale data, and so can normally operate with relaxed requirements for coherence. The second case includes specific circumstances in which the decision to

accept incoherence is made at the time of access, because the need for immediate access to data outweighs the risk of possible incoherence.

Relaxed requirements

For some applications, reading stale data is not an error. For example, in normal operation, a user does not care about executing the latest version of a system utility program, provided the version used is not "too old." Similarly, when reading a computer bulletin board, it is acceptable if articles do not appear immediately, as long as they appear in a reasonably timely manner. Support for incoherent access to replicated data, to increase availability or performance or to reduce costs, has been considered for information retrieval systems (*e.g.*, [2, 29]), as have relaxed serializability constraints for transactions (*e.g.*, [19, 23, 27]). Alonso, *et al.*, [2] propose a number of criteria for the distance allowed between a copy and the true version of the data. The most practical of their proposals is in terms of the maximum time since the data was current.

It is important to note that it is the application, not the data, that determines when incoherent access is allowed. The same data when accessed for different purposes can have different requirements: for example, a user perusing stock prices is satisfied with recent prices, but when making a trade the same user demands current information. The bound on how stale data can be needs to be specified on each request. In V's file service, this could be done by embedding a modifier within the name: for example, `/storage/any/bin/latex:stale=1h` would indicate that a version up to one hour old is acceptable. The modifier need not be obtrusive; including it on directories in the search path for program loading suffices to indicate that a slightly stale version of a system program can be used.

A cache that uses leasing can easily support stale reads alongside coherent access. The only change required is to the condition the client checks before returning cached data: instead of requiring an unexpired lease, it requires a lease that expired not more that T_{stale} seconds ago, where T_{stale} is the bound specified by the application. A default of $T_{stale} = 0$ provides coherent access to those applications that require it.

Accepting stale reads increases availability because data remains readable for a limited period without requiring communication with the server. The length of that period is the bound on how stale data can be and still be acceptable. With a long enough bound, then, accepting stale reads can keep data available for reading across brief communication outages or server crashes. This approach holds promise, but only for those applications

in which the acceptability of stale data can be identified in advance. Applications that normally require coherent access do not benefit.

Specific circumstances

A user might not be willing to sacrifice coherence for continued access to files until he is faced with the file server being unavailable. For example, when a user attempts to read a file in order to begin editing it, the read can fail with an indication that the server is not available. Faced with this result, the user might prefer to proceed by reading the cached copy of the file, instead of waiting for the server to return to operation. It is possible that the user does not require a guarantee of coherence from the cache, since he may be able to determine from the file's contents or other knowledge that the cached copy is current, or at least very likely to be.

The cache manager can help the user decide how to proceed by making available information about the contents of the cache. In addition to showing what data is present, the cache manager should also indicate when each item was last known to be current. The prototype in Chapter 3 could be easily extended for this purpose with an additional tree of names, rooted at `/cache/`*hostname*`/contents`. Each directory within this tree would contain only those names for which data is caches, and each directory entry would consist of a set of flags indicating which of the possible data is present[8] and the time at which the lease covering it expires (or expired). A user can determine both what is in the cache and how stale it might be by listing these directories.

Two extensions would increase the number of situations in which the cache can be used to provide access to otherwise unavailable data. The first extension is to support writing to the cache in addition to reading from it, so as to support a wider range of activities. The writes clearly cannot be committed until the file server is again accessible, but the risk of loss may still be low enough for some uses.

The other possible extension would lower the risk from incoherent access by detecting it when communication with the file server has been restored, and providing the opportunity to resolve (or correct) any conflicting operations. Because sharing is rare, conflicts would also be rare, so that corrective action would seldom be required.

Handling conflicts is very important, though, when data is written to the cache, because a conflict can prevent writes from committing. For example, a user might write to

[8]In the prototype, the fragments of data are the stable attributes in the descriptor, the volatile attributes, and the file's contents or the list if the directory's children.

a cached copy of a file while unable to communicate with the server, and after communication is restored, the cache attempts to commit the writes to the server. If another user, though, has changed permissions for the file, the first user might no longer be authorized to write it, so that the write-back must fail. Fully exploiting caching for increasing availability would require developing techniques to detect conflicts and tools to resolve them, which are beyond the scope of this dissertation.

4.4 Summary

This chapter has broadened the scope of results from previous chapters. First, coherence for secondary caches, which improve the scalability of the system, is easily provided using subleasing. Second, support for atomic transactions does not negate the performance benefit of caching, and leasing can be used to provide efficient concurrency control. Finally, the approaches taken here to caching are applicable when higher availability is needed; leasing is compatible with replicating data at file servers for greater availability, and the coherence of caches can be traded off to make data available even when file servers are not accessible. Taken together, these results strongly suggest that caching—particularly with leasing—can be expected to work well in a wider range of circumstances than those considered in earlier chapters.

Chapter 5

Related work

This chapter surveys related work in four areas:

- maintaining coherence in contexts other than file service,
- measurements of file-system access patterns,
- other caching file systems, and
- uses of time that are similar to leasing.

5.1 Coherence

The problem of coherence arises in several contexts other than file caching: shared memory multiprocessors, distributed shared memory, replicated data, and distributed name service.

5.1.1 Multiprocessors

Shared-memory multiprocessors use caches to reduce contention for the common bus and memory. A variety of protocols have been implemented or proposed, some for implementation in hardware, others for a combination of hardware and software. Descriptions of several of these protocols can be found in [3] and [65]. The focus of multiprocessor work has been on details other than notification, such as the representation of cache directories and evaluation of write-invalidate versus write-broadcast approaches. The coherence mechanisms use reliable notification, since the designs do not attempt to tolerate partial failures.

Several projects are designing multiprocessors intended to accommodate hundreds or thousands of processors (*e.g.* [15]). As such systems grow larger, the ability to tolerate partial failures will become more important, but research on caching in multiprocessors has given little attention to partial failures.

5.1.2 Distributed memory

A shared memory can be implemented even when the underlying hardware does not support it. Li and Hudak [41, 42], for example, describe providing shared memory among processes on different workstations connected by a local-area network. The techniques used depend on reliable notification, and so do not tolerate crashes or communication failures. Leasing could be used in this context, using the extension for write-back caches described in Section 2.4.3. The access pattern and allowable delays differ from those of file service; further evaluation would be required to determine performance.

In the Mirage [24] distributed shared memory, reliable notification is augmented with a timer, but for a different purpose from that in leasing. In Mirage the time specifies a minimum period for a page to remain resident at a site before it is released; the interval can be increased to reduce thrashing on a write-shared page. Within the framework of leasing, this amounts to having a minimum time after acquiring or extending a lease before the leaseholder is willing to relinquish the lease or approve an update.

5.1.3 Distributed naming

Time-based methods resembling leasing have also been used in at least two distributed naming systems. Lampson describes a global directory service [39] in which client caches discard entries at a server-specified time. Servers are forbidden from modifying an entry before it expires. This condition is equivalent to our policy for leases over installed files. Lampson makes no provision, however, for requesting approval of updates or for extending the term for already cached data.

Name services more commonly use cached data as *hints*, for which coherence need not be guaranteed, since stale data can be detected when it is used. In the Internet Domain Name Service [47], for example, a name server specifies a *time-to-live* for the data it returns, and clients cache the data for that period. The data may be modified during that interval, however, and any inconsistency that results must be detected and corrected by other means. Terry [62, 63] discusses in more detail the caching of hints for name

interpretation, including the use of on-use and periodic checks as options in maintaining the accuracy of the cache at the desired level. Using hints shifts the cost of ensuring that data is correct to each access, with no restrictions or load imposed on updates; hints can perform very well when the cost of verifying a hint is small. Leases, in contrast, limit the cost per access to checking whether the lease has expired and extending it when it expires, but shift the burden partially to updates, which must obtain approvals or wait for leases to expire.

5.1.4 Replicated data

As previously noted, caches are replicas, and so techniques for maintaining the mutual consistency of replicas can be used for cache coherence. For example, Gifford's description of weighted voting [28] proposes treating caches as *weak representatives* with no votes, which has the same behavior as leasing with a zero term. Any vote assignment that assigns a read quorum to a cache is equivalent to an infinite-term lease, since the cache must approve every update. Leases can be viewed as temporarily assigned votes, but with much simpler reassignment than in more general schemes for adjusting quorums [8, 34].

Most other algorithms for replication are coherent only with respect to communication via the file service; *i.e.*, operations may appear to be performed out-of-order if there users or application programs exchange communicate by means other than reading and writing files. In contrast, leasing and voting methods make the stronger guarantee of coherence with respect to arbitrary communication, including communication via channels outside the computing system.

Furthermore, available copies algorithms, including virtual partition, are poorly suited for the conditions presented by large-scale caching. In particular, each change to the set of available copies incurs significant overhead. With caching, though, the set of replicas for a particular file changes often as clients cache new files and discard others from their caches. Supporting a large number of clients compounds the problem, since it increases the frequency of changes to the set of accessible replicas. In contrast, leasing is designed to efficiently handle frequent change in the set of clients.

	Ousterhout	Kent	Thompson	Floyd	Burrows	Chapter 3
naming	no	no	some	yes	yes	yes
failed operations	no	no	no	no	yes	yes
descriptor access	no	no	no	no	yes	yes
file lifetimes	yes	no	yes	yes	no	yes
program loading	some	no	no	yes	no	yes
file classes	no	no	yes	yes	some	yes
long-term trace	yes	yes	yes	yes	yes	no
sharing	yes	yes	yes	yes	yes	no

Table 5.1: Data included in trace studies.

5.2 File access patterns

There have been a number of previous studies of file access. Most of the measurements reported have been collected on centralized Unix systems in academic environments [11, 25, 26, 37, 48, 53, 65], but there have also been measurements on IBM mainframes [55, 61] and on Multics [49]. Our comparison here focuses on the Unix measurements, since they are more directly comparable to those collected in V, and they include more relevant information.

Most of the Unix studies collected traces of file-system access over periods of several days. These trace studies are:

- Ousterhout, *et al.*, [53] with additional simulation results in [50]
- Kent [37]
- Thompson [65]
- Floyd [25, 26]
- Burrows [11]

Table 5.1 summarizes the data included in each of these plus the traces from Chapter 3. Mogul [48] reports counts of operations over a period of several days.

Three of these studies, Ousterhout, Kent, and Thompson, focus on access to file contents. All three use the traces to drive simulations of block-level caching to explore effects of block size, cache size, read-ahead, write policy, and coherence. Thompson's analysis is the most thorough and is the only one of these three to include operations other

than reads and writes in the traffic analyzed. Burrows reports on a simple simulation of whole-file caching used as an initial test of the feasibility of caching.

Most of the studies report data on sharing among users; the exceptions are Kent's, which reports on simultaneous sharing among processes, and ours, which includes data from only one user. A study of sharing in Multics [49] describes sharing among user login sessions.[1] The caching simulations place each user on a separate workstation. They all find simultaneous write-sharing to be rare; sequential sharing, in which a file is written by one user and eventually read by some other, is fairly common. Floyd and the Multics study provide additional data about files that are shared. A small number of widely-shared files account for a large portion of the read traffic; these files would be included in our installed class. Also, temporary files are almost never shared. Most sharing of non-installed files is among small numbers of users; the exceptions are identified by Burrows and Thompson as files used in the interface to system services.

Our measurements attempt to give a more complete picture of a client's file-system access than just reads and writes; only our traces include all three of failed operations, descriptor access, and program loading. Each of these three is shown to be a significant share of the load whenever it is measured. Mogul and Burrows both find descriptor reads to be common, and Burrows also reports that 18% of all name lookups fail. Kent also traces disk accesses, and he notes that half of the disk I/O is not accounted for by file reads or writes, but is the result of paging or directory and descriptor access.

In summary, the measurements presented here differ from the other studies in their focus. The traces in V do not attempt to capture either long-term patterns or sharing among users that the other studies describe. Instead of focusing on access to file contents, Chapter 3 looks at the full set of operations requested by clients. Also, the analysis here explores the different effects of caching on classes of files more fully than do the other studies.

5.3 Caching file systems

Several other distributed file systems employing caching have been built. This section looks at three of them in some detail, and compares the issues that they address and the techniques they use with those in this dissertation.

[1] In the terminology of Multics, it reports sharing of segments among processes.

5.3.1 Sprite

Sprite's file system [50] uses RAM to cache file contents on both servers and clients. Sprite is similar to the basic cache in Chapter 3 in that it makes no attempt to cache naming or descriptor information. Nelson estimates that traffic could be reduced by up to half if it did cache that information [51]. Sprite differs from the basic cache in two important ways: its use of delayed write and its coherence mechanism.

Sprite makes use of delayed write on all files in order to reduce both traffic and response time, sacrificing reliability for improved performance. This contrasts with our special handling for temporaries and reliable commit for all others.

Though Sprite makes no distinctions in operation, Nelson [51] does report measurements with writes delayed for only temporary files. When nontemporary files are written through on close, network traffic is 70–109% of that for a thirty second delay on all files. Nelson's measurements of elapsed time are difficult to interpret, since they reflect a disk organization that handles writes very inefficiently: writing one file block often requires writing two disk blocks, with no attempt made to avoid a seek between them. For server policies providing some insulation from this defect, elapsed time with only temporaries delayed is within 5% of that when all files are delayed. In Sprite the creation, deletion, opens and closes for temporary files are still being handled by the file server; performance would be even better under our design, which handles those operations within the cache.

Sprite provides coherence with respect to arbitrary communication, but at the granularity of individual read and write system calls instead of file open and close as in our design. The mechanism is a hybrid, querying the file server on each open and using reliable notification while a file is open. To avoid thrashing, client caching is disabled when a file is concurrently write-shared. Because Sprite depends on reliable notification, it does not tolerate communication failures.

The information for maintaining coherence is kept in the server's RAM; so it is lost when the server crashes. Welch [67] describes the protocol by which a server attempts to recover this state by broadcasting a query to clients. This protocol does not tolerate communication failures, and during recovery a malicious client can interfere with the guarantee of coherence to other clients.

5.3.2 Andrew

The Andrew file system (AFS) is a shared file service intended to augment the local file system of a workstation. There have been two major versions of the Andrew file

system [35, 58], which share the property of caching whole files on local disk. Coherence is provided at the granularity of file open and close, as in V, though in Andrew the choice was primarily made to simplify implementation.

The first version, AFS-1 [58], is equivalent to the basic cache, with all naming performed by the servers, and a coherence check on each open. The servers in AFS-1 had inadequate capacity, for two primary reasons: a high level of traffic, with 65% of the requests for coherence checks and 27% to read descriptors, and an inefficient implementation of the servers.

The second version, AFS-2 [35], made several changes to reduce the load on servers. The file servers were reimplemented in a much more efficient manner, and caching of directories and descriptors was added, with reliable notification used to maintain coherence. Cached directories are interpreted by clients to look up names. (The representation of directories was also changed to make lookups much more efficient.)

The use of reliable notification means that AFS-2 cannot tolerate communication failures. When a server's attempt to notify a cache about an update fails, based on the transport-level timeout, the server discards its record that the client had the file or directory cached and then proceeds with the write [36]. This leaves stale data in the client's cache, and the client learns of the error only when it next attempts to communicate with the server. During the interval that it is using stale data, a client may continue to read and write files from other servers. To limit the duration of inconsistencies, each client queries its servers every ten minutes to synchronize its clock.

There is some class-specific handling of files under AFS. AFS is intended to augment a local file system; temporaries and copies of some installed files are placed in the local file system instead of AFS to reduce the demand. This also means that those files cannot be shared. Also, installed files are placed in separate read-only volumes to allow them to be replicated and to eliminate the need to maintain records of clients using them. Updates to these volumes must be made through a different mechanism from that used for writable files.

5.3.3 MFS and Echo

Burrows' MFS [11] was developed concurrently with this work. It seeks to provide coherent access to a shared Unix file system with a similar degree of reliability. Data is cached on local disk and in the existing Unix buffer cache, using the Unix file-system data structures. The NFS protocol [57] is used to access the file servers, with a separate *token*

server used to ensure coherence. MFS uses the thirty-second delay on writes normally provided by the Unix kernel, and so can suffer undetected loss of recently written data when a client crashes.

The tokens in MFS are very similar to leases for write-back caching, but without the emphasis on time that is central to leasing. Because a token is revoked when the server's RPC to the client times out, the token scheme does not tolerate communication failures, and the guarantee provided is comparable to that of AFS-2. Burrows does mention the possibility of timing out tokens, but does not discuss the tradeoffs in selecting the term. MFS is able to specify tokens at the granularity of individual bytes within files, and Burrows describes efficient server data structures for maintaining them. Security is discussed in more detail than it is here, since Byzantine clients are a more serious problem when attempting to recover with write-back caches.

The Echo file system [45] uses the same approach to coherence as MFS, but with communication failures handled correctly. Echo uses a primary-site approach to replication, with lease information replicated at the secondaries [44]. Its designers discuss several options for extending leases, but the trade-offs with length of term are not quantified. They also do not consider the special handling that we provide for installed files.

5.3.4 Others

The Cedar file system CFS [60] avoids the problem of maintaining coherence by limiting sharing to immutable files. Because shared data cannot change, no coherence mechanism is required. Instead, selection of versions to cache is made by an application that establishes a name space local to the workstation. Caching of immutable data can also be expressed as a special case of leasing with an infinite term, which in this case is acceptable since writes are prohibited.

RFS [7] is similar to Sprite, using a version check on each open and disabling caching when concurrent write-sharing is detected. The major differences from Sprite are that caches are write-through, that caching is enabled more quickly after write-sharing ends. No description of failure-handling is provided.

Coda [59] extends AFS-2 with support for replication and for *disconnected operation* when the file servers are not available. Coda does not guarantee coherence of either caches or replicas in the presence of communication failures, yet a workstation continues to access files in the cache and on other servers without giving any indication that a failure has occurred. Only some inconsistencies are detected after the fact; no effort

is made to check for read-write conflicts. As a consequence, many applications (such as Unix make) cannot be safely run on top of the Coda file system. Coda's approach stands in contrast with that described in Chapter 4, which makes any use of stale data safe. Coda does include tools for resolving conflicts that it does detect, and for caching a desired set of files before intentionally disconnecting a portable host from the network.

The FileNet system [20] employs caching for data with an access pattern very different from the other systems considered here. The rate of access is very low, with the rate of reads measured being around one per ten seconds, and the lowest measured ratio of reads to writes is greater than forty; in addition, files shift between periods of read-only access and periods of update. To minimize server traffic, the FileNet system tracks the rates of reads and writes for each file and switches between the equivalent of leasing with terms of zero and infinity. The formula used for deciding when to switch is similar to the estimates for traffic developed here.

5.4 Other uses of time

Leasing's use of time is far from unique: timeout is a standard technique in communication protocols for detecting lost messages, and timeouts are often the only means available for detecting host or process crashes in distributed systems, including database systems. Some database systems use timeouts to detect deadlocks as well [1]. Leasing differs from timeout, though, in that it uses time not so much to detect failures as to make guarantees in spite of them. In this respect, leasing is similar to protocols such as SCMP [43] that use either explicit time or bounded packet lifetime in order to suppress duplicates.

5.5 Summary

This research differs from previous work in its focus, and the resulting cache design is distinguished by the techniques used to improve its performance.

The first difference in focus is the insistence on robustness: caching is not allowed to compromise the coherence, reliability, or availability of the file service. Furthermore, communication failures must be tolerated, not just host crashes.

The other difference in focus is the treatment of file service as a whole, instead of limiting interest to reading and writing file contents. This broader concern manifests

itself in the caching of naming and descriptor information in the prototype and in the extensions in Chapter 4 to support transactions, concurrency control, and replication.

The cache design has two distinctive features. It ensures coherence by leasing, with its emphasis on the explicit use of time. The design also distinguishes two classes of files, temporary and installed, and exploits their differences to improve performance.

Chapter 6

Conclusion

6.1 Results

This dissertation has shown how to use caching of file-system data on clients to improve the scalability and performance of a distributed file service without reducing its ability to tolerate either host or communications failures. Analysis of the performance of a prototype file-service cache for the V-system shows that it achieves a substantial reduction in server traffic without decreasing the file system's reliability, availability, or coherence. Three techniques make possible this combination of performance and fault-tolerance: using leasing for coherence, recognizing file classes, and caching metadata as well as file contents.

Leasing. Leasing guarantees coherence with respect to arbitrary communication, even in the face of host crashes and communication failures. Furthermore, a faulty or even malicious client cannot compromise the coherence of other clients' caches. The only effect that one client's failure has on another client is that writes to shared files might be delayed for up to the term of a lease. Instead of depending on reliable communication, leasing requires only that the client and server have reasonably well-behaved clocks.

The mechanisms previously used for file cache coherence can be expressed as special cases of leasing, where the term is of either infinite or zero duration. Leasing makes the use of time explicit, which allows a range of trade-offs between normal performance and worst-case performance after a failure. For the patterns of access that have been measured in file systems, an analytical model shows that leases with terms of just a few seconds make the contribution of coherence to both traffic and response time very small, while also providing a reasonable bound on the added delay after a client failure.

Trace-driven simulation confirms this result, and the model indicates that performance remains good as processor speed or network latency increases.

Leasing is also flexible. The conditions for correctness allow a variety of policies for managing leases, making different trade-offs between traffic and response time or read and write performance. Maintaining the coherence of multi-level caches through subleasing supports scaling to very large systems. Leasing is also compatible with replicating data at multiple servers, and it can work together with a framework that, when acceptable, allows access to possibly inconsistent data in order to increase availability. Finally, leasing can be used within a transaction-processing system to allow caches to perform concurrency control.

File classes. Not all files are the same, and recognizing the differences for just a few classes of files offers substantial improvements in performance. Recognizing installed files and using a different policy for managing the leases over them can reduce the traffic per client and avoid pathological behavior on the updates to them, at the expense of increasing the latency for those infrequent updates. Similarly, providing additional support for temporary data reduces traffic for writes and for file creation and deletion; the overall reduction in traffic compares favorably with that achieved by delaying for up to thirty seconds writes to all files. Unlike delayed write, though, special handling for temporaries reduces traffic without sacrificing the reliability of storage for non-temporary files.

Caching metadata. File service includes more than just access to the contents of files. Once file contents have been cached, a significant fraction of the traffic handled by a file server reads information about files. For the traces analyzed in Chapter 3, caching naming information and descriptors makes possible a reduction in traffic of up to 40% compared to caching only file contents, though that benefit also depends on the use of leasing to avoid coherence checks. A detail that has escaped previous research is the significance of caching to handle failed name lookups.

Together, these techniques yield a traffic ratio of 13–50%, an improvement of as much as 60% compared to a basic cache that does not use them. Some features of the design presented in Chapter 3 are specific to the V-system, as are details of the analysis of performance. Similar results, however, would be expected for other systems, especially with a similar workload.

6.2 Future research

Coherence. Chapter 2 noted the difficulty in defining precisely what is meant by coherence. The notion of coherence *with respect to* a set of possible observations needs to be formalized more completely and, if possible, unified with the notions of transaction serializability and of different levels of coherence in multiprocessor memory systems.

Also, the usefulness and performance of leasing needs exploration in other applications of caching, such as distributed shared memory or scalable multiprocessors, where failure-handling has not yet received much attention.

Sharing in distributed systems. The evaluation of cache performance would be much more solid with data on the sharing that actually does occur in distributed systems. Traces of file access in distributed systems would be very useful, especially if made in a system that allows users to easily employ multiple hosts. The scheduling of programs in such a system also needs to be reconsidered in light of caching.

No measurements have been published for access patterns in wide-area file systems, but such data is needed in order to evaluate the effectiveness of multi-level caching in such a setting.

File server design and performance. Cache performance is only one factor in the overall performance of a file service; the performance of the file server is also very important. Caching not only reduces the average demand placed on the server by each client, but it changes the nature of that load as well. With effective caching, the file server's load is dominated by writes, reversing the conventional wisdom that reads should be optimized at the expense of writes. Response time is especially dependent on the latency of commit operations. The performance of server design alternatives needs to be reevaluated in light of caching. Of particular interest are the different uses of logging described by Finlayson [22], by Hagmann [33], and by Ousterhout and Douglis [52], and the use of different storage technologies, such as nonvolatile RAM, disk arrays, and write-once disks.

Caching in other contexts. Finally, research is needed to determine the potential for caching in other storage services, such as databases, in which the techniques from Chapter 4 could be used to support caching and transactions together. The possibility of caching in an object-oriented database is especially attractive, though access to relational

databases within a distributed environment could also benefit.

Appendix A

Full trace data

This appendix provides more detailed information about the traces used in Chapter 3.

A.1 Configuration traced

All three traces described here were collected on a MicroVAX II workstation with sixteen megabytes of memory; several different remote file servers were used. The caching server used for tracing corresponds to the basic cache plus temporary support. All traces were collected with the workstation otherwise idle, except for normal system overhead; the file servers were under their normal loads. Each trace begins with the cache completely empty.

The traces were collected without kernel support for virtual memory, using a process-level memory server for block-level caching. The absence of virtual memory affects the measurements in three ways. First, execution is slightly slower, because the overhead for the process-level server is higher than that for a kernel memory server, in both context switching and in copying of data. Second, program loading generates a higher level of reads than would be expected from demand paging. Third, the process-level memory server used for the traces transfers whole files only, which inflates the number of reads and the cache size. None of these significantly affects the issues considered in this dissertation. Partial transfers would yield a slightly lower traffic ratio for reads than is reported in Chapter 3, which can only increase the significance of the improvements described there.

A.2 Data included

The traces include a record for each operation handled by the caching server, and for each operation it requested of the file server. Reads and writes are not traced, but each close record contains the number of one-kilobyte blocks read from or written to the cache. The operations within each group reported in Chapter 3 are:

commit These operations all commit an update to the file system.

- **open - create** Create a new file by opening it for writing.
- **close - commit** Commit writes to a file.
- **create directory** Create a new directory.
- **remove/rename** Remove or rename a file.
- **write named descriptor** Modify the descriptor for a file or directory, usually to change permissions.

naming read This group of operations all require resolving a name or other reading of descriptors, but produce no changes.

- **open - read** Open a file for reading.
- **reopen - write** Open an existing file for writing.
- **get file name** Get the name of an open file. In V, this is invoked as part of the program loading sequence.
- **read descriptor** Read the descriptor of an already-open file.
- **read named descriptor** Read a descriptor by name.
- **directory open** Open a directory for reading.
- **directory read** Read a single entry from a directory.
- **name not found** Operations that failed because the named file or directory did not exist. Mostly opens for reading and descriptor reads.

misc. The remaining operations neither require reading descriptors nor commit a visible change.

- **truncate** Truncate an open file.

close - no change Close a file without committing any writes, usually because it was open only for reading.

directory close Close a directory after reading it.

Some operations are not included in the trace; their use was determined or estimated from the data that is included. Truncate and get-file-name are not traced at all, but their use could be reliably determined by examining the traces. For directory reading, only the open is recorded, and the number of entries read was determined by examining the directories after the trace completed.

The trace also does not include counts of blocks read from or written to the file server; these counts are estimated from the counts of blocks accessed from the cache. In the activities traced, most blocks that are accessed are read or written only once for each time the file is opened; so that the number of distinct blocks accessed from an open file is very close to the number of block accesses. Because no cache replacement is required, the number of blocks read or written from the server is the maximum number of blocks read or written for a single open at the cache. For a few cases in which blocks from a file are read repeatedly, the number of blocks in the file is used as the number of blocks read from the server.

A.3 The data

The data for each trace appears in five tables.

Traffic with no caching. These counts are of the operations that applications requested of the cache, which would all be handled by the file server if there were no cache.

Traffic for basic cache. Only reads, writes, opens and closes change when the basic cache is used, and on-open coherence checks are added. Counts of blocks read and written by the file server are not included in the traces, but can be determined from the file sizes, since the memory server used for the traces does only whole-file transfers.

Write traffic for a thirty-second delay is based on lifetimes in the traces, measured from the time a version is committed to the time it was overwritten or deleted. Note that this underestimates traffic for a typical implementation of delayed write, in which any dirty blocks are written out every thirty seconds: the average delay in such an

implementation is fifteen seconds. The comparison in Section 3.3 is therefore biased in favor of delayed write, yet the special support for temporaries compares favorably.

Traffic with descriptor caching. The data reported for descriptor caching is based on a simple simulation in which the entire directory is read when a name is not found or when an application reads the directory. Coherence is still checked on each use of descriptor data except for the reads within a directory.

Traffic with special temporary support. These counts are determined directly from the traces. The table is omitted for `latex`, because it uses not temporary files.

Lease extensions. The figures reported here are from a separate simulation. The first column reports on traffic if extensions are requested separately as needed, the second if an extension is piggybacked on each committing operation.

Counts of extensions are shown for installed and other files both separately and if combined; the totals are significantly lower when combined. Temporary files are not included, since the special support for them eliminates the need for coherence.

operation	tmp	installed	other
read (kB)	13872	42265	17178
write (kB)	11596	0	9641
truncate	0	0	0
open - read	338	3030	1901
open - create	340	0	72
reopen - write	0	0	53
close - no change	338	3030	1901
close - commit	340	0	125
create directory	0	0	3
remove/rename	340	0	17
get file name	0	401	0
read descriptor	0	7	170
read named descriptor	0	192	629
write named descriptor	0	0	26
directory open	0	0	0
directory read	0	0	0
directory close	0	0	0
name not found	0	340	4236

Table A.1: `fsbuild`: Traffic with no caching.

operation	tmp	installed	other
read (kB)	0	2763	512
write (kB)	11596	0	9641
with 30-sec. delay	2284	0	7700
truncate	0	0	0
open - read	0	94	78
open - create	340	0	72
reopen - write	0	0	0
close - no change	338	94	78
close - commit	340	0	125
coherence check	338	2936	1876
max. cache size			9 MB

Table A.2: `fsbuild`: Traffic for basic cache.

operation	tmp	installed	other
get file name	0	0	0
read descriptor	0	1	8
read named descriptor	0	5	107
write named descriptor	0	0	26
directory open	0	4	10
directory read	0	173	155
directory close	0	4	10
name not found	0	4	10
coherence check	338	3862	6776

Table A.3: `fsbuild`: Traffic for directory/descriptor caching.

operation	tmp
read (kB)	0
write (kB)	0
truncate	336
open - read	0
open - create	4
reopen - write	0
close - no change	4
close - commit	0
remove/rename	4
coherence check	0

Table A.4: `fsbuild`: Traffic for special temporary support.

files included	separate extensions	piggybacked extensions
installed	398	398
other	373	233
combined	508	388

Table A.5: `fsbuild`: Lease extensions required, 10-second term.

operation	tmp	installed	other
read (kB)	2625	17858	4542
write (kB)	1773	0	2035
truncate	0	0	0
open - read	70	480	402
open - create	74	0	102
reopen - write	0	0	2
close - no change	70	480	402
close - commit	74	0	104
create directory	0	0	20
remove/rename	74	0	6
get file name	0	476	0
read descriptor	0	1	27
read named descriptor	0	0	258
write named descriptor	0	0	22
directory open	0	0	96
directory read	0	0	793
directory close	0	0	96
name not found	0	0	725

Table A.6: afsbench: Traffic with no caching.

operation	tmp	installed	other
read (kB)	0	1093	402
write (kB)	1773	0	2035
with 30-sec. delay	545	0	1611
truncate	0	0	0
open - read	0	24	71
open - create	74	0	102
reopen - write	0	0	0
close - no change	70	24	173
close - commit	74	0	104
coherence check	70	456	333
max. cache size			5 MB

Table A.7: afsbench: Traffic for basic cache.

operation	tmp	installed	other
get file name	0	0	0
read descriptor	0	1	4
read named descriptor	0	0	22
write named descriptor	0	0	22
directory open	0	0	32
directory read	0	0	151
directory close	0	0	32
name not found	0	0	11
coherence check	70	932	1370

Table A.8: `afsbench`: Traffic for directory/descriptor caching.

operation	tmp
read (kB)	0
write (kB)	0
truncate	71
open - read	0
open - create	3
reopen - write	0
close - no change	3
close - commit	0
remove/rename	3
coherence check	0

Table A.9: `afsbench`: Traffic for special temporary support.

files included	separate extensions	piggybacked extensions
installed	96	96
other	107	49
combined	123	67

Table A.10: `afsbench`: Lease extensions required, 10-second term.

operation	tmp	installed	other
read (kB)		1732	114
write (kB)		0	124
truncate		0	0
open - read		70	20
open - create		0	0
reopen - write		0	8
close - no change		70	20
close - commit		0	8
remove/rename		0	0
create directory		0	0
get file name		2	0
read descriptor		0	0
read named descriptor		0	0
write named descriptor		0	0
directory open		0	0
directory read		0	0
directory close		0	0
name not found		0	82

Table A.11: `latex`: Traffic with no caching.

operation	tmp	installed	other
read (kB)		820	52
write (kB)		0	124
with 30-sec. delay		0	124
truncate		0	0
open - read		10	6
open - create		0	0
reopen - write		0	4
close - no change		10	6
close - commit		0	8
coherence check		60	18
max. cache size			3 MB

Table A.12: `latex`: Traffic for basic cache.

operation	tmp	installed	other
get file name		0	0
read descriptor		0	0
read named descriptor		0	0
write named descriptor		0	0
directory open		0	2
directory read		0	68
directory close		0	2
name not found		0	2
coherence check		62	96

Table A.13: `latex`: Traffic for directory/descriptor caching.

files included	separate extensions	piggybacked extensions
installed	12	12
other	17	13
combined	17	13

Table A.14: `latex`: Lease extensions required, 10-second term.

Bibliography

[1] R. Agrawal, M.J. Carey, and L.W. McVoy.

The performance of alternative strategies for dealing with deadlocks in database management systems.

IEEE Transactions on Software Engineering, SE-13(12):1348–1363, December 1987.

[2] Rafael Alonso, Daniel Barbara, Hector Garcia-Molina, and Soraya Abad.

Quasi-copies: Efficient data sharing for information retrieval systems.

In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology — EDBT '88*, number 303 in Lecture Notes in Computer Science, pages 443–468, Venice, Italy, March 1988. Springer-Verlag.

[3] James Archibald and Jean-Loup Baer.

Cache coherence protocols: Evaluation using a multiprocessor simulation model.

ACM Transactions on Computer Systems, 4(4):273–298, November 1986.

[4] Association for Computing Machinery.

Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, December 1985.

Published as *Operating Systems Review*, 19(5).

[5] Association for Computing Machinery.

Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing, Calgary, Alberta, August 1986.

[6] Association for Computing Machinery.

Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona, December 1989.

Published as *Operating Systems Review*, 23(5).

[7] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh.

A remote-file cache for RFS.

In *Proceedings of the Summer 1987 Usenix Conference*, pages 273–279, Phoenix, Arizona, June 1987. Usenix Association.

[8] Daniel Barbara, Hector Garcia-Molina, and Annemarie Spauster.

Increasing availability under mutual exclusion constraints with dynamic vote reassignment.

ACM Transactions on Computer Systems, 7(4):394–426, November 1989.

[9] Eric J. Berglund.

An introduction to the V-System.

IEEE Micro, pages 35–52, August 1986.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman.

Concurrency Control and Recovery in Database Systems.

Addison-Wesley, 1987.

[11] Michael Burrows.

Efficient data sharing.

Technical Report No. 153, Computer Laboratory, University of Cambridge, December 1988.

The author's Ph.D. thesis.

[12] L.M. Censier and P. Feautrier.

A new solution to coherence problems in multicache systems.

IEEE Transactions on Computers, C-27(12):1112–1118, December 1978.

[13] David R. Cheriton.

UIO: A uniform I/O system interface for distributed systems.

ACM Transactions on Computer Systems, 5(1):12–46, February 1987.

[14] David R. Cheriton.

The V distributed system.

Communications of the ACM, 31(3):314–333, March 1988.

[15] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle.

Multi-level shared caching techniques for scalability in VMP-MC.

In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 16–24. ACM SIGARCH, IEEE Computer Society, May 1989.

[16] David R. Cheriton and Timothy P. Mann.

Decentralizing a global naming service for improved performance and fault tolerance.

ACM Transactions on Computer Systems, 7(2):147–183, May 1989.

[17] David R. Cheriton and Willy Zwaenepoel.

Distributed process groups in the V kernel.

ACM Transactions on Computer Systems, 3(2):77–107, May 1985.

[18] Peter B. Danzig.

Finite buffers and fast multicast.

In *Proceedings of the 1989 SIGMETRICS and PERFORMANCE '89 International Conference on Measurement and Modeling of Computer Systems*, pages 108–117. ACM SIGMETRICS and IFIP WG 7.3, May 1989.

Also published as *Performance Evaluation Review* 17(1).

[19] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen.

Consistency in partitioned networks.

ACM Computing Surveys, 17(3):341–370, September 1985.

[20] David A. Edwards and Martin S. McKendry.

Exploiting read-mostly workloads in the FileNet file system.

In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* [6], pages 58–70.

[21] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.

The notions of consistency and predicate locks in a database system.

Communications of the ACM, 19(11):624–633, November 1976.

[22] Ross Stuart Finalyson.

A log file service exploiting write-once storage.

Technical Report STAN-CS-89-1272, Stanford University, Department of Computer Science, July 1989.

The author's Ph.D. thesis.

[23] M.J. Fischer and A. Michael.

Sacrificing serializability to attain high availability of data in an unreliable network.

In *Proceedings of the ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 70–75, New York, May 1982. Association for Computing Machinery.

[24] Brett D. Fleisch and Gerald J. Popek.

Mirage: A coherent distributed shared memory design.

In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* [6], pages 211–223.

[25] Rick Floyd.

Directory reference patterns in a UNIX environment.

Technical Report TR 179, University of Rochester, Department of Computer Science, August 1986.

[26] Rick Floyd.

Short-term file reference patterns in a UNIX environment.

Technical Report TR 177, University of Rochester, Department of Computer Science, March 1986.

[27] Hector Garcia-Molina and Gio Wiederhold.

Read-only transactions in a distributed database.

ACM Transactions on Database Systems, 7(2):209–234, June 1982.

[28] David K. Gifford.

Weighted voting for replicated data.

In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, California, December 1979. Association for Computing Machinery.

Published as *Operating Systems Review*, 13(5).

[29] H.M. Gladney.

Data replicas in distributed information services.

ACM Transactions on Database Systems, 14(1):75–97, March 1989.

[30] Cary G. Gray and David R. Cheriton.

Leases: An efficient fault-tolerant mechanism for distributed file cache consistency.

In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* [6], pages 202–210.

[31] James N. Gray.

Notes on database operating systems.

In R. Bayer, R.M. Graham, and F. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1979.

[32] J.N. Gray, R.A. Lorie, G.R. Putsolu, and I.L Traiger.

Granularity of locks and degrees of consistency in a shared data base.

In G.M. Nijssen, editor, *Modelling in Data Base Management Systems*, pages 365–394. North-Holland, 1976.

[33] Robert Hagmann.

Reimplementing the Cedar file system using logging and group commit.

In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, Texas, December 1987. Association for Computing Machinery.

Published as *Operating Systems Review*, 21(5).

[34] Maurice P. Herlihy.

Dynamic quorum adjustment for partitioned data.

ACM Transactions on Database Systems, 12(2):170–194, June 1987.

[35] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West.

Scale and performance in a distributed file system.

ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[36] Michael Leon Kazar.

Synchronization and caching issues in the Andrew file system.

Technical Report CMU-ITC-058, Information Technology Center, Carnegie Mellon University, June 1987.

[37] Christopher A. Kent.

Cache coherence in distributed systems.

Research Report 87/4, Digital Equipment Corporation Western Research Laboratory, December 1987.

A slightly revised version of the author's Ph.D. thesis.

[38] Leslie Lamport.

Time, clocks, and the ordering of events in a distributed system.

Communications of the ACM, 21(7):558–565, July 1978.

[39] Butler W. Lampson.

Designing a global name service.

In *Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing* [5], pages 1–10.

[40] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel.

File access performance of diskless workstations.

ACM Transactions on Computer Systems, 4(3):238–268, August 1986.

[41] Kai Li and Paul Hudak.

Memory coherence in shared virtual memory systems.

In *Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing* [5], pages 229–239.

[42] Kai Li and Paul Hudak.

Memory coherence in shared virtual memory systems.

ACM Transactions on Computer Systems, 7(4):321–359, November 1989.

[43] Barbara Liskov, Liuba Shrira, and John Wroclawski.

Efficient at-most-once messages based on synchronized clocks.

In *Proceedings of SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 41–49. Association for Computing Machinery, September 1990.

Published as *Computer Communications Review*, 20(4).

[44] Timothy Mann.

Personal communication (electronic mail), July 1989.

[45] Timothy Mann, Andy Hisgen, and Garret Swart.

An algorithm for data replication.

Research Report 46, Digital Equipment Corporation Systems Research Center, June 1989.

[46] Timothy Paul Mann.

Decentralized naming in distributed computer systems.

Technical Report STAN-CS-87-1179, Stanford University, Department of Computer Science, September 1987.

The author's Ph.D. thesis.

[47] P. Mockapetris.

Domain names — concepts and facilities.

Request for Comments 1034, Network Information Center, SRI International, Menlo Park, CA, November 1987.

[48] Jeffrey Clifford Mogul.

Representing information about files.

Technical Report STAN-CS-86-1103, Stanford University, Department of Computer Science, March 1986.

The author's Ph.D. thesis.

[49] Warren A. Montgomery.

Measurements of sharing in MULTICS.

In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 85–90, West Lafayette, Indiana, November 1977. Association for Computing Machinery.

Published as *Operating Systems Review*, 11(5).

[50] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout.

Caching in the Sprite network file system.

ACM Transactions on Computer Systems, 6(1):134–154, February 1988.

[51] Michael Newell Nelson.

Physical memory management in a network operating system.

Technical Report UCB/CSD 88/471, Computer Science Division (EECS), University of California, November 1988.

The author's Ph.D. thesis.

[52] John Ousterhout and Fred Douglis.

Beating the I/O bottleneck: A case for log-structured file systems.

Operating Systems Review, 23(1):11–28, January 1989.

[53] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson.

A trace-driven analysis of the UNIX 4.2 BSD file system.

In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* [4], pages 15–24.

[54] David A. Patterson, Garth Gibson, and Randy H. Katz.

A case for redundant arrays of inexpensive disks (RAID).

In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, June 1988. ACM SIGMOD.

[55] J. M. Porcar.

File migration in distributed computer systems.

PROGRES Report 82.6, University of California at Berkeley, Electronics Research Laboratory, July 1982.

The author's Ph.D. thesis.

[56] Daniel R. Ries and Michael Stonebraker.

Locking granularity revisited.

ACM Transactions on Database Systems, 4(2):210–227, June 1979.

[57] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.

Design and implementation of the Sun network filesystem.

In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, Portland, Oregon, June 1985. Usenix Association.

[58] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West.

The ITC distributed file system: Principles and design.

In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* [4], pages 35–50.

[59] Mahdev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere.

Coda: A highly available file system for a distributed workstation environment.

IEEE Transactions on Computers, 39(4):447–459, April 1990.

[60] Michael D. Schroeder, David K. Gifford, and Roger M. Needham.

A caching file system for a programmer's workstation.

In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* [4], pages 25–34.

[61] Alan Jay Smith.

Disk cache—miss ratio analysis and design considerations.

ACM Transactions on Computer Systems, 3(3):161–203, August 1985.

[62] Douglas B. Terry.

Caching hints in distributed systems.

IEEE Transactions on Software Engineering, SE-13(1):48–54, January 1987.

[63] Douglas Brian Terry.

Distributed name servers: Naming and caching in large distributed computing environments.

Technical Report UCB/CSD 85/228, Computer Science Division (EECS), University of California, March 1985.

The author's Ph.D. thesis.

[64] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton.

Preemptable remote execution facilities for the V-System.

In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* [4], pages 2–12.

[65] James Gordon Thompson.

Efficient analysis of caching systems.

Technical Report UCB/CSD 87/374, Computer Science Division (EECS), University of California, October 1987.

The author's Ph.D. thesis.

[66] Ken Thompson.

UNIX implementation.

The Bell System Technical Journal, 57(6):1931–1946, July–August 1978.

[67] Brent Ballinger Welch.

Naming, state management, and user-level extensions in the Sprite distributed file system.

Technical Report UCB/CSD 90/567, Computer Science Division (EECS), University of California, April 1990.

The author's Ph.D. thesis.

Reproduced by NTIS